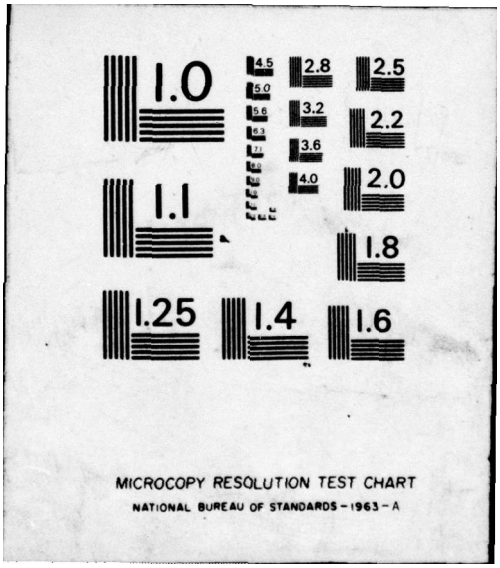
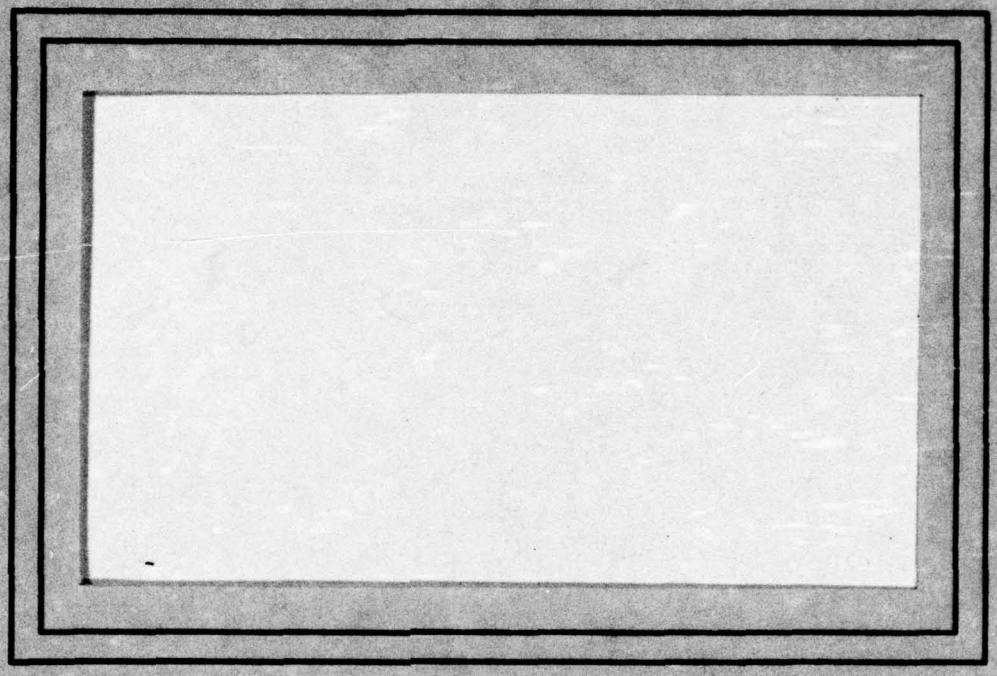
AD-A032 812 MARYLAND UNIV COLLEGE PARK DEPT OF COMPUTER SCIENCE F/G 12/1
SPONTANEOUS COMPUTATION IN COGNITIVE MODELS.(U)
JUL 76 C RIEGER N00014-76-C-0477
UNCLASSIFIED TR-459 NL

1 OF 2
AD A032812


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS - 1963 - A



COMPUTER SCIENCE TECHNICAL REPORT SERIES

UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742

(14) TR-459
(15) N00014-76C-0477
NR 049-386

(12) 115 p.

(11) Jul 76

(6) SPONTANEOUS COMPUTATION IN COGNITIVE MODELS

(10) Chuck Rieger
Department of Computer Science
University of Maryland
College Park, Maryland 20742

(9) Technical rpt.,

ABSTRACT: The engineering and theory of a style of computation in which code runs spontaneously (as opposed to on demand) are developed. The notion of a spontaneous computation (SC) is defined, briefly surveyed, and compared to other styles of computation. Then, in the first half of the paper, a LISP-based system which carries out a general theory of SC is described. This includes: complex trigger patterns, organization of SC trigger patterns into associative "trigger trees", and the structure of an SC itself. Higher level organization and control of SC are then discussed, introducing the notion of a "channel". In the second half of the paper, some theoretical ideas about how to use SC in cognitive models, particularly those modeling language comprehension and problem solving, are presented and discussed. The discussion includes: SC as a model of non-algorithmic inference, SCs as "character followers" in a story comprehension system, SCs as subgoal protectors and plan optimizers in a problem solver, and the relationships among SC, context and frames. In particular, ideas related to partially triggered SCs, and their theoretical applications as context-focusers and motivation-generators are explored. The paper represents one aspect of a larger project called the Commonsense Algorithm Project, and includes as appendices a self-contained system of LISP code which implements many of the ideas discussed in the text.

The research described in this report was funded by the Office of Naval Research under contract number N00014-76C-0477.

(15)

CONTENTS

SPONTANEOUS COMPUTATION IN COGNITIVE MODELS

Chuck Rieger
Department of Computer Science
University of Maryland
College Park, Maryland 20742

1. INTRODUCTION

The computations in any model of intelligence can be classified into two categories: those which are invoked on demand, and those which occur spontaneously. A demand-based computation is one which occurs in response to an explicit request for a service or for information, i.e., a call by name or a call by pattern. Spontaneous computation on the other hand is computation which is unsolicited; it simply happens in reaction to some condition or set of conditions becoming true. As such, spontaneous computation will represent the associative component of any model. It may either interrupt a demand-based computation, or it may serve to initiate or augment a demand-based computation.

In models of human intelligence, generic examples of demand-based computation are problem solving, where a goal is stated then solved, and deduction, where a question is posed then proved or disproved according to some logical framework. An example of spontaneous computation in cognitive models is inference, where there is no demand source, but where new information is nevertheless derived from existing information.

Spontaneous computation typically reacts to states of a data base, or to changes in a data base (as in MICROPLANNER [SWC1] and CONNIVER [MS1]), but in general might react to arbitrary states of computation, including calling sequences in certain contexts, and so forth.

This paper is about the general theory and practice of spontaneous computation (hereafter abbreviated SC), addressed from within the framework of LISP-based models of intelligence such as are being developed under the style of modeling known as Artificial Intelligence. In the paper, we will address

these major issues:

(1) How ought spontaneous computation to be implemented?

(2) How can it be harnessed in fruitful ways?

(3) What are its theoretical uses in modeling human intelligence?

(4) How can SC be coordinated with demand computations; i.e., when and how do spontaneous computations interact with demand-based computation?

(5) What is the relationship of SC to existing theories of intelligence; that is, what theories might have SC as their bases?

The paper is intended to be an assessment of some state of the art ideas about spontaneous computation, together with some (hopefully new) ideas about the engineering and theoretical utility of such computation in modeling human intelligence.

1.1 Background

There is considerable uncertainty concerning the roles of demand-based computations (which we will sometimes call "doers"), and spontaneous computations (which we will sometimes call "watchers") in models of human intelligence, especially with regard to (a) what classes of cognitive processes each is best suited for, and (b) how doers and watchers interact and are coordinated.

The use of SC in models of intelligence in the past has been extensive, but its theoretical role remains largely a mystery, since its theoretical applications have been piecemeal.

MICROPLANNER [SWC1], based on Hewitt's PLANNER [H1], was probably the first programming language based principally upon the notion of spontaneous computation. In MICROPLANNER, changes to a central data base are monitored by a population of watchers, called THANTE and THERASING "theorems". As any given pattern is entered or deleted from the database, some subset of this population can react, i.e., gain control of the computation after the pattern has entered or left the database. Computations thus triggered, being

arbitrary LISP programs in form, can then do whatever they desire. The once-radical aspect of such a programming style was of course that more might happen at run-time than would meet the eye by a simple inspection of the demand-based flow of control evident in the source listing!

Since MICROPLANNER, CONNIVER [MS1], and a host of other languages with similar features have evolved with SC as a major component (e.g. QA4 [R4], QLISP/INTERLISP [RS1,T1], POPLER [D1]). Also, there has been considerable interest in so-called "production" systems, where computation is phrased as a collection of rewrite rules which, in some systems, can be quite sophisticated. (See Davis and King [DK2] for a good overview; see also Tesler et. al. for LISP70 [T2], and Newell and Simon [NS1].) There are many similarities between production based systems and spontaneous computation as we approach it here, because in both paradigms some sort of pattern matching lies at the base of all computation invocations.

However, MICROPLANNER and production style languages have provided only the framework for SC; they do not define its theoretical roles in cognitive modeling. One of the first cognitive theories which relied heavily on SC was Charniak's model of children's story comprehension [C1]. Charniak's thesis was that understanding long segments of meaningful, connected language is primarily a matter of planting watchers in response to clues found at one point in the story, hoping that later on in the story some of those watchers will spring to life at the correct moment to provide an interpretation for what is happening, and do so in a way that causes interpretation to be a function of what has come before in the story. In other words, Charniak employed SC as the basis of a sow/reap, prediction/fulfillment mechanism.

Charniak ran into an understandable combinatorial explosion of sorts; although his idea was a good one, we feel this approach constituted a basic theoretical misuse (overuse at least) of spontaneous computation. Later in the paper, we will make an attempt to delimit - at least abstractly - the types of presumed human mental processes for which SC seems most appropriate.

More recently, Marcus [M1] has employed the notion of SC as a model of English grammar, where rules of syntax are formulated as procedures which run spontaneously whenever their run conditions are matched by some portion of the contents of the sentence buffer of the sentence under analysis. Marcus

incorporates a notion of "packets" of SCs to control which subpopulation of SCs it is that is allowed to react at any given moment.

Although the theoretical merit of this approach over other more conventional control structures for grammar is not yet certain, Marcus apparently does not experience combinatorial explosions, and is able to express grammar in very modular and clever (if sometimes run-time obscure!) form.

Still more recently, the KRL group (Bobrow, Winograd, et al.) have incorporated a notion of "procedural attachment" into their system which is being designed as a base language for expressing cognitive models [BW1]. Procedural attachment is a specific form of spontaneous computation that lies closer to demand-based computation than most. The MIT LISP machine [G1] also uses a very similar notion of procedural attachment. We will have a brief look at this idea later on.

1.2 The CSA Theory and Spontaneous Computation

Up to this point in our own research, we have been interested in processes which are most properly classified as doers. The research, called The Commonsense Algorithm Project, has so far been concerned mainly with the development of a representation for commonsense cause and effect knowledge, and the development of an organization which permits us to store and access large numbers of so-called CSA patterns in useful ways.

The CSA theory has been an attempt to unify some ideas about language comprehension and problem solving; it is described in [R1], [R2] and [R3]. Because this unification has been our goal, the doers in the existing CSA model are, abstractly speaking, twofold:

(1) The plan synthesizer, which, given an agent and a goal (expressed via the set of state and statechange predicates known to the system), will construct a plan (i.e. build a novel CSA pattern up from its knowledge store of smaller patterns) which could be employed by the agent to accomplish the goal; and

(2) The language interpreter, which, given a situation and an action, will search "backwards" through the knowledge base of CSA patterns

and arrive at a most reasonable interpretation (i.e. reason) for the action in the situation. Interpretations are thus sensitive to the context defined by the situation; because of this, we feel we have the kernel of a reasonably powerful story comprehender.

The central theme in these segments of the CSA research has been that intelligent selection is the basic issue with regard to demand-based computation. The general statement CSA is making is that doers must have good reasons for solving a given problem, or answering a given question in the way they decide to do it. Our point of view is that intelligent selection at every step where selection is possible is a necessary (and almost sufficient!) cornerstone of all human intelligence. We feel that selection is perhaps the key issue of demand-based computations, and that the development of computational paradigms in which selection is made central will provide the foundations for research into human learning. [R1], [R2] and [R3] present this point of view and give descriptions of specific aspects of the existing CSA theory.

However, it was recognized early in the CSA research that doers are only half of the model - that SC is an equally important aspect of some CSA ideas. It is only recently that we have gotten into the business of SC, and our interest was initially motivated by a facet of CSA having to do with mechanism description and simulation [RG1].

In the CSA representation, it is also possible to capture what we call the "causal topology" of man-made devices. Causal topology means a description of the cause and effect relationships implied by the structure, and evident in the operation of a man-made device; this topology can be purely physical, or a mixture of social and physical descriptions. We are presently able, for a large class or mechanisms, to write a CSA pattern which captures the internal workings of a device, express this pattern in a form suitable for communication with the computer model, and store it in memory, integrating it into the set of existing patterns for other mechanisms and commonsense algorithm patterns used by the plan synthesizer.

Since, among others, one of our goals was to be able to use the CSA pattern describing an arbitrary device as the basis of a mechanism simulator

(which in turn is scheduled to become the heart of a CAI "Mechanisms Laboratory"), we were confronted with the design of a simulator. Inspired by Sussman's work with electronic circuit analysis using a simple SC basis [SS1], we adopted the following strategy for our simulator: convert the CSA pattern which describes a mechanism to a population of SC-based procedures, each of which models one local aspect of the flow of causality within the mechanism, then light the fuse by presenting the population with a starting pattern, and watch it go!

The mechanisms simulator is now running [RG1], and represents the first fully developed application of the CSA SC component (to be described), which itself has been under development for the past several months.

However, since the initial ideas about the simulator, we have grown more interested in SC for its own sake, and as the basis of certain classes of inference in story comprehension (described later in the paper). Also, we have grown more and more interested in the nature of interactions between demand-based computation and spontaneous computation, specifically those between "watchers" and the doers in the existing CSA model.

The main questions are:

(a) What is the division of labor between these two computational paradigms, and

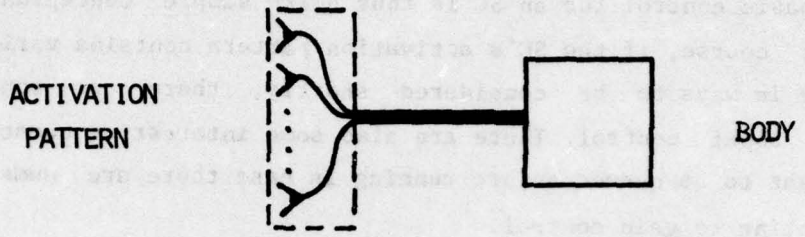
(b) How do demand-based computations and spontaneous computation constructively coexist and cooperate?

Hence, the remainder of the paper will be about a CSA-independent theory of spontaneous computation, but motivated from within the specific CSA framework. The first part is design and engineering; the second part is theory.

1.3 SC Basics

Any SC has two parts: an activation pattern and a body. The activation pattern is a description of the situation or class of situations to which the SC will react, i.e. spontaneously run itself. The body is the computation it performs, most generally an unrestricted LISP computation (although there may be good reasons in theory for restricting its form).

An SC is thus pictured as:



Display 1.

To illustrate these two components, consider an SC which embodied a naive model of earth gravity. Simply put, earth gravity tells us that, as long as we are close to the earth, whenever an object is unsupported it will begin moving toward the earth. We would express the activation pattern as:

```
(UNSUPPORTED X)          (object X is unsupported)
AND
(DISTANCE X EARTH RELSMALL)   (object X and the earth
                               are relatively close)
```

and the body as:

```
ASSERT: (STATECHANGE LOCATION X EARTH)
(assert that X is moving toward the earth)
```

Display 2. Earth gravity.

Of course, this example of an SC is a bit oversimplified, but it gives a beginning insight into how and why one might employ SCs. In this case, such a pattern which could run spontaneously when its activation pattern occurred would be of use in a problem solving system: it could answer "what-if"

questions, and it could interrupt the demand-based problem solver whenever it has left an object in an unsupported condition, or it could reconstruct the probable location of an object which had been left unsupported, and so on.

The basic control for an SC is thus quite simple conceptually observe-react. Of course, if the SC's activation pattern contains variables or if it is complex in ways to be considered shortly, there are some interesting questions about control. There are also some interesting questions about how the SC ought to be queued before running in case there are numerous SCs who are attempting to gain control.

1.4 Central Arenas

In the larger picture of an entire system using this style of computation, spontaneous computation implies the existence of a possibly large population of processes which lurk on the edge of a "central arena" of computation being performed by populations of doers. The metaphor is one of a fishbowl, so we will elevate this word to the status of para-technical term, and imagine that some demand-based computation is "fishbowled" by populations of watchers who can kibbitz, modify, augment or interrupt the main computation.

These metaphors lead to some interesting questions:

(1) What is the nature of the central arena that SCs watch?

(2) What are the watchers eyes made of?

(3) What parts of the central arena are visible to the watchers?

(4) When should a watcher who thinks it sees something of importance be allowed to leap into the central arena (gain control)? and

(5) What ought a watcher be able to do, once in control?

The first half of the remainder of the paper will describe the design and engineering we have done to address these questions within the CSA framework.

2. ENGINEERING SPONTANEOUS COMPUTATION

2.1 Structure and Organization of SCs

2.1.1 Elementary Trigger Patterns

The eyes of a watcher are commonly called the watcher's trigger pattern. A trigger pattern is a description of some aspect of the state of a computation, i.e., a pattern in the central arena (whatever the arena is) to which to react. In principle, a trigger pattern could range in complexity from a zero/one signal on a computer interrupt vector up to an instantaneous description of every particle in the universe!

Since we are approaching the theory from a LISP implementation, a more modest and natural type of trigger pattern for us is a LISP S-expression, or generalized list structure. Since S-expressions are the medium in which all computations are based in LISP, we will have a fairly general SC trigger pattern definition.**

** This definition will permit access to all "user level" aspects of LISP-based computation. However, it will provide no means by which to base SCs upon LISP's background control, e.g. LISP's internal stacks and other mechanisms. We will later discuss schemes in which SCs which react to this level of computation can be expressed.

While, in principle, unrestricted LISP S-expressions would be the most desirable substrate for trigger patterns, we have developed a storage technique which is most naturally applied to a mildly restricted class of LISP S-expressions and will provide nearly the same expressive power. This is the class of nested n-tuples, where a nested n-tuple <nn> is defined as:

<nn> := <constant> | <variable> | (<nn> ... <nn>)

<constant> := <LISP atom>

<variable> := -<LISP atom> (read as a hyphen sign)

Display 3. Definition of a nested n-tuple.

Nested n-tuples built up in this fashion (and then into more powerful constructions) will permit us to express patterns which we can regard as associative, in the sense that they will have the potential for "matching", or "triggering on" (to be defined) patterns of activity in the central arena.

To illustrate, suppose we wish to construct a relatively simple spontaneous computation whose eye, or trigger pattern would react to arena patterns of the form: "(wake me up when) Someone knows that John loves someone." Then we might write:

(KNOWS -X (LOVES JOHN -Y)) **

Display 4. Someone knows that John loves someone.

as this SC's trigger pattern. Here, by our convention (there are many other reasonable conventions), -X is a variable, -Y is a variable, and the rest of the pattern is constant.

** The particular choice of representation (i.e. predicates and form) is of course up to the theorist; we are interested here mainly in the structure of the patterns he employs.

The interpretation of the pattern of Display 4 would be: activate the body of the SC to which this pattern is attached as trigger pattern whenever a pattern which matches this pattern occurs in the central arena, communicating to the body the identities of the symbols which were bound to the variables -X and -Y by the match process. For example, the pattern

(KNOWS MARY (LOVES JOHN RITA))

Display 5. A fact which matches the pattern of Display 4.

would match this pattern, setting -X to MARY, -Y to RITA.

Now, there are some new questions:

(1) What is the interpretation of variables?

(2) What kinds of expressive power built up from this simple definition would be desirable in trigger patterns?

(3) How ought the system to store a trigger pattern so that it can be accessed associatively in an efficient manner from among a large population of SCs?

2.1.2 Trigger Pattern Variables

A variable in an SC trigger pattern is essentially a stand-in for a "don't-care" part of the trigger pattern. However, since variables can be regarded as unique individuals (i.e., by using a different name for each variable in the pattern), they can serve also to restrict the class of objects to which they may be bound, and hence the class of arena patterns to which the trigger will react.

The simplest convention for variable restriction is that variables with the same name must be bound to the same object when a match to the pattern is being attempted. For example, if we choose to limit the applicability of the trigger pattern of Display 4 to only those cases where the person who KNOWs is the same person as the person John LOVEs, we might express this as:

(KNOWS -X (LOVES JOHN -X))

Display 6.

Of course, if we choose this convention, either the pattern matcher or the inherent organization of the trigger pattern associative access mechanism (or both) must be able to deal with this type of restriction.

But this kind of restriction, which we will call variable identity restriction, can actually be regarded as a special case of a more powerful scheme which would allow us to attach arbitrary restrictions on a variable. Such restrictions could be denoted, e.g., by the form:

(<variable> : <restriction> ... <restriction>)

Display 7. Hypothetical variable restriction syntax.

---

where each <restriction> would contain references to <variable>. Then, for example, we could further constrain the trigger pattern about loving in Display 6 to read: "Someone who hates John knows that John loves his (the hater's) sister" by writing:

```
(KNOWS (-X : (HATES -X JOHN))
       (LOVES JOHN (-Y : (SISTER-OF -Y -X))))
```

Display 8.

Of course, our syntax would need some refining (e.g. to clarify the status of (HATES -X JOHN), indicating that it is a query to the system database, rather than a general LISP computation, and so forth).

Variable identity restriction is a case of this more general scheme, which we will call variable semantic restriction, because writing -X in more than one place is logically equivalent to using variables of different names, but qualifying them so that they must all be lisp EQUAL:

(LOVES -X -X) <=> (LOVES -X (-Y : (EQUAL -Y -X))

Display 9. Identity vs. semantic restriction.

(This is assuming that -X will receive a binding prior to -Y in the match process).

However, there is another logically equivalent way to approach variable semantic restrictions which will become evident when we consider the extensions to the basic trigger facility in the next section; so we will not pursue the notion further here.

2.1.3 Complex Trigger Patterns

In PLANNER and CONNIVER, trigger patterns are kept simple by permitting no more than one logical part, and by adopting only forms of the variable identity restriction technique. Because of this, in MICROPLANNER and CONNIVER, a trigger pattern will provide an SC with only superficial evidence that the SC is applicable to an arena situation. Any detailed applicability tests must be performed as part of the SC's body, even though they are logically part of the trigger condition.

While this approach provides a powerful enough basis for general SC, it would be more desirable to incorporate increased expressive power in the trigger. Doing so will amount to moving into the trigger (where it belongs) computation which would otherwise have to be performed by the body of the SC after becoming active on the basis of a superficial trigger condition.

In one sense, where the triggering intelligence resides - in the trigger pattern itself, or in the SC's body as a set of additional preconditions to be tested prior to the actual running of the logic of the body proper - is simply a matter of style. However, there are two arguments in favor of moving more of the triggering intelligence into the trigger pattern:

(1) Conceptually, the trigger and the body have nothing to do with each other, except for the passing of variable bindings from the

trigger to the body at invocation time. If it ever becomes of theoretical interest to have the system manipulate its own SCs' trigger patterns and bodies (say, for "learning"), keeping these two components totally distinct has obvious advantages.

(2) Computationally, when a population of simple, PLANNER-like watchers becomes large and many watchers appear to be relevant to the current situation in the central arena, it may be the case that many of them are not actually ready to run, but are requesting to run only to make further relevance tests. From such tests, they might discover themselves to be irrelevant after all. On the other hand, other SCs might actually be ready to run and perform some useful computation. This poses a problem to the higher level arbiters in the system who are trying to decide whom to run, and it also presents potential complications to some invocation processes which split trigger patterns apart, as described later in the paper.

For these reasons, we have adopted the view that it is desirable to move intelligence into the trigger pattern, with the intent of providing more total conceptual and computational isolation between the trigger and the body.

Let us now consider the types of expressive power we will and will not need in trigger patterns.

2.1.3.1 Ellipsis: Too Costly

There are many conceivable ways to increase the expressive power of SC triggers. One obvious extension would be to allow for ellipsis in the patterns, regarding them more as strings than as list structures. For example, it might be convenient to write trigger patterns such as:

(LOVES ...)

and (... JOHN ... MARY ...)

Display 10.

---

meaning "match anything that starts with the constant LOVES" and "match

anything that contains the symbols JOHN and MARY in it (in that order)", respectively.

While ellipsis might be desirable in principle, we feel that it incurs more problems in practice than it is worth in the context of a LISP environment. For one thing, since ellipses can represent zero or more unspecified objects, accomodating ellipsis introduces a combinatorial element into what is otherwise a straightforward (linear) pattern matcher.

Beyond this, if we are writing trigger patterns within an environment where some theory of representation is providing a general semantic framework for expressing knowledge, then the utility of ellipsis is not nearly so great as it is in unrestricted string environments. For example, if we regard LOVES as a predicate, with an intentional semantics (as well as an extensional semantics) of its own rather that simply as a string of five symbols, and if we adhere to a particular syntax for using it (say, Cambridge Polish notation), then ellipsis will often stand for something which could have been specified more succinctly without ellipsis anyway, e.g.:

(LOVES ...) <=> (LOVES -X -Y)

Display 11. Ellipsis in a system with intentional semantics.

So, although there are clearly some cases where it would be convenient to have ellipsis, we have chosen to reject ellipsis as an extension to the trigger pattern expressive power.

2.1.3.2 Unordered Sets: Not a Big Problem

Another possible extension would be to provide for the expression and matching of unordered sets in the trigger. Although, wherever a predicate with intensional semantics is involved we can use a positional structure, there are cases where we may wish to deal with sets of things whose order is truly of no consequence. Symmetric predicates and sets whose membership can be anticipated, but whose order cannot, are cases in point. For example, we might wish to write a trigger pattern which would be sensitive to the notion "Bill

is a brother of John", and it would be convenient to have to write only:

(BROTHERS BILL JOHN)

Display 12. Symmetric predicates.

---

and let some other part of the system worry about the symmetric nature of this predicate. Or perhaps we would like to react to any pattern in which Mary sees John, Bill and Pete all at the same time:

(SEE MARY (JOHN BILL PETE))

Display 13. Unordered sets.

---

again, deferring the problems in matching the unordered set (JOHN BILL PETE) with some particular ordering to some other part of the system.

The associative access structure for SC trigger patterns we have developed and will describe will not (conveniently) tolerate this type of generalization. However, we believe we do not lose any significant expressive power, since, once again, it is usually possible to reformulate unordered set matching within an ordered framework:

(SEES MARY (JOHN BILL PETE))

(SEES MARY -X) and (EQUAL-SETS -X '(JOHN BILL PETE))

Display 14.

---

where we attach (still in the trigger pattern) the additional restriction on -X that it match a particular unordered set. The unordered set match would be carried out by calling a special LISP function, say EQUAL-SETS, to accomplish the match.

Now that we have described two things we have not done, we will describe what we have done.

2.1.3.3 What We Did: Complex CSA Trigger Patterns

In addition to some of the generalizations suggested by abstract pattern matching needs, we will want SC trigger patterns to be of sufficient expressive power for the types of theoretical uses to which we will wish to put SCs. Most important in this respect will be a flexibility in describing subtle or complex situations; if triggers are too coarse, then we run the risk of having too many SCs wake up in any given situation. Therefore, it will be essential that the SC trigger pattern conventions hamper the theorist as little as possible when the need arises to express trigger patterns for matching complex goings-on in the central arena.

Everything we want in an SC trigger pattern seems to converge on one rather simple theme: allow the trigger pattern to be composed of numerous logical parts, connected via logical predicates. Allow each logical part to be either

(a) a nested n-tuple pattern which will represent an associative component of the complex trigger pattern, or

(b) an arbitrary restriction, formulated as a general LISP function call.

These two components will fill both the need to react spontaneously (the structural pattern part), and the need to perform more subtle checking prior to the actual invocation of the associated SC.

We will call any general LISP function call which exists within a trigger pattern a computable of the trigger pattern. We will want the associative components of a trigger pattern to become indexed into the system-wide control for spontaneous computations, whereas we will want the computables simply to be retained in the pattern to be used at associate-time, but not indexed into the SC control.

A definition of a more general SC trigger pattern syntax, <tp>, which follows from these ideas is:

```
<tp> := <assoc> | <computable> | <complex>
<assoc> := (+ <effort> <nn>)
<computable> := <LISP-S-expression>
<complex> :=   (AND <tp> ... <tp>)
             | (OR <tp> ... <tp>)
             | (ANY <tp> ... <tp>)
```

Display 15. The CSA complex trigger pattern syntax.

where <nn> is a nested n-tuple as defined in Display 3, and where it is understood that <LISP-S-expression> includes all forms except those covered by <assoc> and <complex> forms.

The semantics of this definition will be as follows: whenever we wish to designate an associative part of a complex trigger pattern (i.e., one which will react to activity in the central arena, and which will come to be indexed into the associative SC access mechanisms to be described), we will encase it with the special marker "+", as in:

```
(+ 1 (LOVES -X MARY))
```

Display 16. A complex trigger associative component.

Alternatively, any portions of the trigger pattern not so encased will be interpreted as LISP computables.

Complex patterns of associative parts and computable parts may be built from the relations AND, OR and ANY. The semantics of an AND condition are that all its parts must be bound in a consistent way in order for the AND relation to be true. In an OR relation, at least one of the OR components must be true, and any variables which come to be bound will reflect the bindings of the first component of the OR found to be true. The ANY relation provides a way of forcing as many of the ANY components as possible to be sought (and, hence, as many bindings as possible to be made); ANY relations are always logically

TRUE. Many other useful relations are imaginable, but we have limited the present system to these three.

As it will turn out, (+ <effort> <nn>) forms will need to interface with both the SC associative control and with the system's deductive and database components (in a fashion to be described). Therefore, this form will also serve as a system-wide query form, i.e., entry syntax into the deductive component of the system. The <effort> field, an S-expression that will evaluate to an integer, will denote the maximum acceptable level of effort to be expended if the form were to be regarded as a database query (i.e. deduction). "Effort" is defined as the number of raw database fetches made.

Often, it will be convenient to include in a complex trigger pattern deductive queries which are identical to + forms in all respects except that they are not to be treated as associative components of the trigger pattern. That is, often we will want to express conditions that must be true before the SCs computation can be activated, but which themselves cannot initiate the running of the SC directly, i.e. trigger it. We adopt the convention of encasing such forms in "-" signs:

(- <effort> <nn>)

Display 17. Non-associative trigger conditions.

Applying this complex trigger syntax to expressing the pattern we considered a while back as Display 8, "Someone who hates John knows that John loves his (the hater's) sister", we can now write:

```
(AND (+ 1 (KNOWS -X (LOVES JOHN -Y)))
     (+ 1 (HATES -X JOHN))
     (+ 1 (SISTER-OF -Y -X)))
```

Display 18.

Here, we have written a trigger pattern all of whose parts are associative. This will mean that the pattern will "nibble" whenever any one of its components is seen in the central arena. The 1's in the <effort> field will indicate that, in the process of attempting to verify that all parts of this pattern are true after any one part has nibbled, only one unit of effort is to be expended for each additional part. (This will amount to restricting the deductive component of the system to one database fetch apiece.)

Note that we now have a vehicle for expressing general forms of variable semantic restrictions: we simply factor out the restrictions and include them as other AND conditions in the pattern.

If we wished to have the trigger of Display 18 react only to the "primary" idea, namely, "Someone knows that John loves someone", then we could simply make the second and third components non-associative:

```
(AND (+ 1 (KNOWS -X (LOVES JOHN -Y)))
     (- 1 (HATES -X JOHN))
     (- 1 (SISTER-OF -Y -X)))
```

Display 19.

To illustrate a pattern which includes computables, we might attempt to capture the notion of "unrequited love with a minor" (!) as:

```
(AND (+ 1 (LOVES -X -Y))
     (+ 1 (NOT LOVES -Y -X)) **
     (- 1 (AGE -X -AGEX))
     (- 1 (AGE -Y -AGEY))
     (#GREATERP -AGEX #18)
     (#LESSP -AGEY #18))
```

Display 20. A complex trigger pattern with computables.

** Negation in the CSA system is handled as shown here, i.e., in patterns themselves rather than in the structures which relate patterns. A future paper will describe the deductive components of the CSA system in more detail.

Here, we have made the notion of unrequited love "central" by specifying only the love/not-love parts of the pattern as associative. The ages of the parties involved are first determined by non-associative database queries, then compared against the integer #18 ** via the computables.

** Because of the way LISP stores them, we cannot index numeric atoms in the ways required by our storage structures. For this reason, we prefix numbers with a # to make them non-numeric. We have a parallel set of arithmetic functions for these forms.

We have described most of the important points of SC trigger patterns as they are now defined and implemented in the CSA system, and feel that we have at least the kernel of a very expressive system. Since we will see more examples of trigger patterns throughout the paper, we will not dwell on their form here, but instead now turn to a despription of the associative access mechanism which is built around this style of complex trigger patterns.

The issues are:

(1) How are the associative components of complex trigger patterns indexed into a central SC control and access mechanisms?

(2) What is the procedure for determining whether a trigger pattern has been matched in its entirety (and hence, whose associated SC body ought to be invoked)?

(3) How ought SCs whose patterns are only <u>partially</u> satisfied to be handled?

(4) What are the interesting theoretical uses to which partially activated SC trigger patterns can be put?

For the remainder of Part I, we will assume that the trigger pattern

definition we have given will be at least necessary for any reasonable SC system, and disregard as best as possible all its shortcomings. This will enable us to investigate some interesting engineering and theoretical questions within a specific framework.

2.1.4 SC Associative Access Paradigm

In any given application of spontaneous computation, we would generally expect there to be large populations of SCs, each SC having the type of complex trigger pattern we have just described. The question therefore is: how ought such populations of trigger patterns to be organized in a system that will need to awaken them associatively?

An SC access requirement is fundamentally different from the access requirement of a database/deductive mechanism. Typically, a database/deductive component is confronted with a pattern which possibly contains some variables. The general type of response from a database/deductive component is a list of ways to fill all the variable positions to make the query pattern "true", i.e. to make it correspond with fully constant patterns in the data base, or with deduced, but still fully constant patterns:

---

accesses

<pattern with variables> --------> list of possible ways to satisfy the pattern

Display 21. Database access paradigm.

For example, if we ask the question "Who are lovers?":

(LOVES -X -Y) ?

Display 22.

---

we would expect a response having the general form: "Facts G123 and G32

indicate, respectively, that (LOVES JOHN MARY) and (LOVES SUE BILL), so I will return the value G123 with -X=JOHN, -Y=MARY, and the value G32 with -X=SUE and -Y=BILL."

The SC component on the other hand will be reacting to patterns which are generally fully constant. It will attempt to locate some subset of the population of trigger patterns (containing variables) which "cover", or match a given fully constant pattern. If we call this subset which associates with a given constant stimulus the "nibblers", the SC paradigm is:

triggers
<fully constant n-tuple> --------> collection of nibblers

Display 23. SC access paradigm.

Although it is possible to use one technique for both database and SC access, it would appear to be more efficient to separate these functions and to engineer the SC component with a technology different from the standard database technologies. (See e.g. McDermott [M2] for a good discussion of database organization). Because of this, we have developed a special technique for SC trigger pattern associative access which is distinct from the standard database access techniques.

2.1.5 Trigger Trees

The organization of CSA SC trigger patterns is based upon a structure we call a trigger tree (sometimes abbreviated TT). A trigger tree is a central structure into which all the associative components of all SC trigger patterns in a given population of SCs can be knit. It will then become possible to speak in terms of "populations of SCs", meaning a tree of triggers, and in terms of "planting" the associative components of an SC's complex trigger in some trigger tree.

"Planting" the associative parts of a complex SC trigger pattern, C, will involve:

(1) fragmenting the complex pattern, C, extracting the list of all its associative components

(2) storing each associative component of C in a specified trigger tree

(3) associating with each associative component of C so planted, say A, list, L(A), of other components of C which, together with A, would cause C to be entirely satisfied.

We will now describe these steps, and the structure of trigger trees. All the activities about to be described are set in motion in the CSA system by calling the function $PLANT:

($PLANT <tp> <sc-body> <tt>)

Display 24. The CSA SC trigger planting function.

---

i.e. "plant in trigger tree <tt> an SC whose trigger pattern is <tp> and whose body is <sc-body>." ** We will describe $PLANT in more detail later.

** We have decided to include as appendices the LISP code which implements the ideas being presented. One could argue about the usefulness of such a decision, since the contribution of any computer system lies not in its code, but rather in what it suggests in the theory. However, much of our programming has been at a level of detail that ought not to have to be repeated by people who are interested in using ideas we describe.

2.1.5.1 Fragmentation

In planting, the complex trigger pattern, composed from AND, OR and ANY relations, computables and associative parts, must first be decomposed into its parts. The decomposition must be performed in a way that couples to each associative component of the pattern a list, L(A), of other parts of the pattern such that if A union L(A) were satisfied, the entire trigger pattern would be satisfied, i.e. fully triggered. This will permit the pattern to be "entered" via any of the associative components, since each associative

component will be planted in an associative access trigger tree and will have a knowledge of which other components of the complex trigger constitute its L(A). Whenever a stimulus causes A to associate, this L(A) will be called up in a more goal-directed mode that tries to determine whether or not A union L(A) has been satisfied.

We will define polling to be that process which, upon some associative component, A, being triggered, tests the remaining pattern components, L(A). Polling is of considerable theoretical interest and will be discussed in more detail later.

The CSA function which, given a trigger pattern <tp>, performs this fragmentation, is called $FRAGMENT and is called initially by ($FRAGMENT <tp> NIL). $FRAGMENT accepts a complex trigger pattern as defined in Display 15 and returns a list of the form:

```
( ( <a1> . <L(a1)> )
  ( <a2> . <L(a2)> )
      .
      .
      .
  ( <an> . <L(an)> ) )
```

Display 25. Fragmentation results.

where <a1> is the first associative component of the complex pattern, <L(a1)> is its associated L(a1), and so forth.

The algorithm is recursive and "understands" the semantics of AND, OR and ANY so that it produces the minimum L(A) for each associative component A in the complex trigger. For example, if $FRAGMENT is called with the trigger pattern

```
(OR (AND (+ 1 (LOVES -X JOHN))
         (OR (+ 1 (LOVES -X BILL))
             (+ 1 (LOVES PETE -X))))
    (+ 1 (NOT LOVES -X JACK)))
```

Display 26.

it will return the list:

```
(((+ 1 (LOVES -X JOHN)) .                    i.e. a1
    (OR (+ 1 (LOVES -X BILL))                     L(a1)
        (+ 1 (LOVES PETE -X))))
 ((+ 1 (LOVES -X BILL)) .                         a2
    (+ 1 (LOVES -X JOHN)))                        L(a2)
 ((+ 1 (LOVES PETE -X)) .                         a3
    (+ 1 (LOVES -X JOHN)))                        L(a3)
 ((+ 1 (NOT LOVES -X JACK)) .                     a4
    NIL))                                         L(a4)
```

Display 27. An example fragmentation.

where multiple references to any one subcomponent of <tp> are preserved as LISP EQ references. Each associative part on the list so produced will then be planted the trigger tree specified in the original call to $PLANT.

Planting an associative part, A, will give rise to a unique path in the trigger tree. At the terminal node of this path we will store a reference to the SC of whose complex trigger pattern A is a part, and attach to this reference L(A) to identify those parts of the original trigger pattern which must be polled in order for the complete pattern to be satisfied in case the complex trigger is entered via A.

We turn now to the structure of a trigger tree.

2.1.5.2 Trigger Tree Structure

Since triggers are associative and, given any fully constant nested n-tuple as stimulus, we want to access all relevant triggers, it would be desirable to have the triggers share as much common storage and structure as possible. This will make it more efficient to access relevant triggers in "parallel" while also conserving space.

A tree is a natural for these purposes (see Knuth [K1], e.g.). Hence, our thinking developed along the line of a tree structure, in which each path from the root of the tree to a terminal node corresponds uniquely to some associative component of some complex trigger. Associative access in this type of structure amounts to a breadth-first traversal of the entire tree, dropping from consideration any paths which fail to match at some point. For any stimulus pattern, P, a traversal of the tree will yield either NIL (no SC associative components nibble at P), or a list of tree terminal nodes representing the subset of the tree's associative patterns that fire in response to P.

The structure of a non-terminal node, <tt-nt-node>, in a trigger tree is: (terminal nodes will be described shortly)

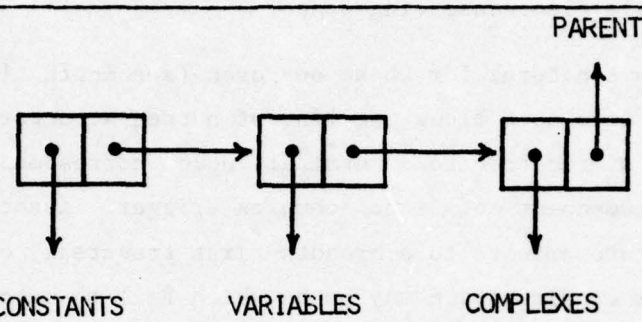
```
<tt-nt-node> :=  (<constants> <variables> <complexes> . <parent>)
<constants> := NIL | (<const-alt> ... <const-alt>)
<const-alt> := (<atom> . <tt-node>)
<variables> := NIL | (<tree-variable> <var-alt> ... <var-alt>)
<var-alt> := (<var-restriction> . <tt-node>)
<var-restriction> := <tree-var> | FREE
<tree-var> := -X <path>
<complexes> := NIL | (<complex-alt> ... <complex-alt>)
<complex-alt> := (<length> . <tt-node>)
<parent> := <tt-node>
```

Display 28. Trigger tree non-terminal node syntax.

---

where a <constant> is any LISP atom which is not an SC variable (an atom

prefixed by a hyphen sign in our convention), <path> is a sequence of integers uniquely identifying the position (to be defined) of the non-terminal node in the tree, and <tt-node> is any trigger tree node. <parent> provides an upward backlink from each node to its parent, with the parent of the root node being NIL. Each trigger tree node thus requires three CONS nodes at a minimum:



Display 29. Trigger tree non-terminal node.

Besides the parent link, which is used only in the process of removing an associative pattern from a trigger tree, each non-terminal node consists of three parts:

(1) constants (indexed by themselves)

(2) variables (indexed by identity restrictions)

(3) complexes (indexed by length)

Each path from the root to a terminal node in the tree will correspond to a left-to-right, depth- first traversal of some associative pattern (a nested n-tuple, possibly containing variables). Each node on any given path in the tree will therefore correspond to some position within the associative pattern which that path represents.

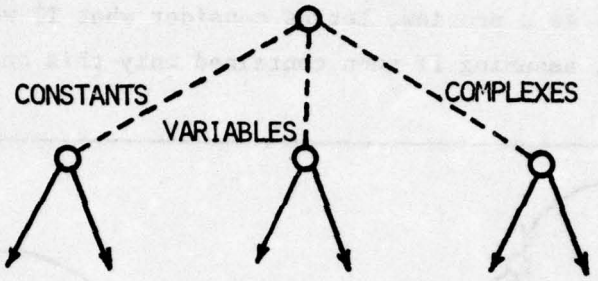
There will be two modes of access for trigger trees:

(1) PLANT MODE: given an associative pattern, find (or create) the path in the pattern tree which represents this trigger, and

(2) ASSOCIATE MODE: given a fully constant stimulus, find all paths to terminal nodes which satisfy this stimulus, i.e. associatively

locate all the nibblers.

We visualize each non-terminal trigger tree node graphically as:



Display 30. Non-terminal node structure.

A <path> to item X in an associative pattern (nested n-tuple) P is defined as the list of "go to position i and descend" operations that would be required in order to move from the entry node of the pattern to X. Rather than define it formally, we will simply illustrate a path. If the pattern P is:

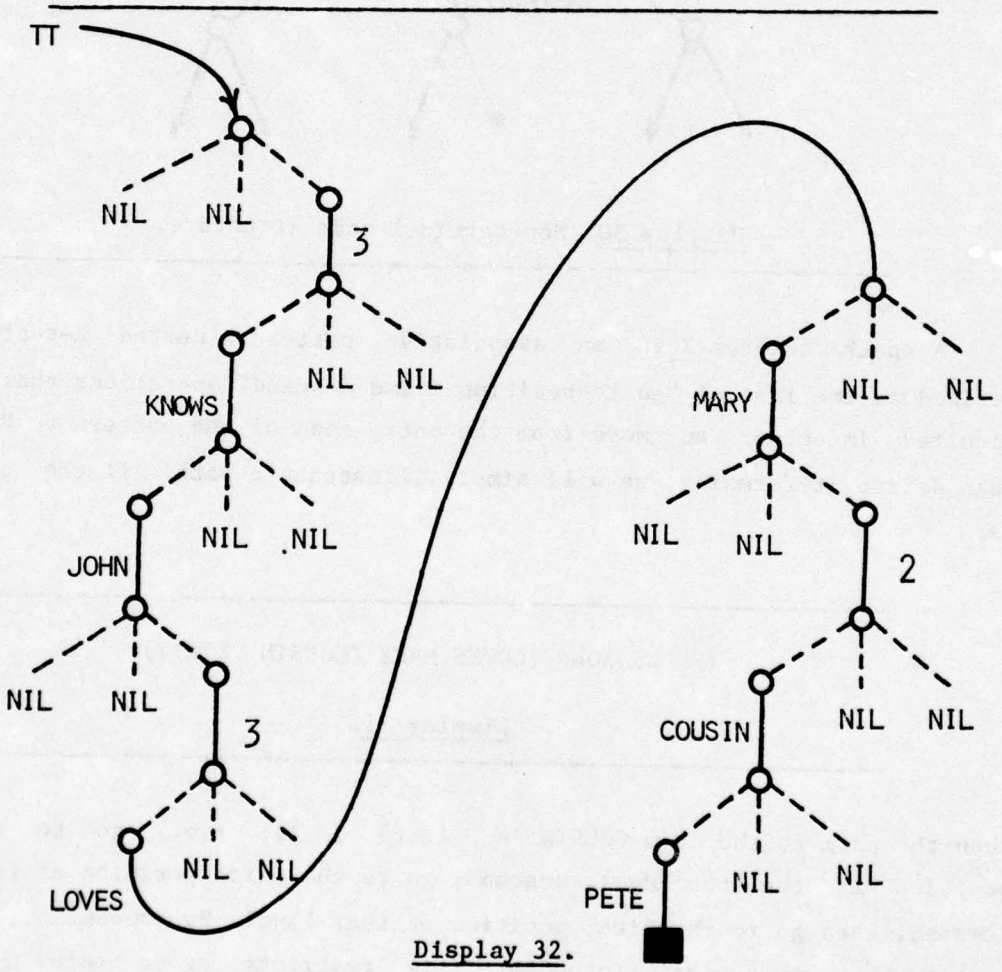
(KNOWS JOHN (LOVES MARY (COUSIN PETE)))

Display 31.

then the path to the atom COUSIN in P is (3 3 1), i.e., go to the third position at the top level, descend, go to the third position at that level, descend, then go to the first position at that level. By convention, we store such paths as decimal integers; this restricts us to nested n-tuples of maximum length about 9 and depth about 10, because of the way integers are stored on the machine. (This is a trivial aspect of the implementation which could be changed easily.)

2.1.5.3 Planting Associative Patterns

Suppose that we wish now to plant the (in this case fully constant) associative pattern, P, of Display 31 in a trigger tree, TT. Starting at the root of TT (if there is no root, or no next node, one will always be created in PLANT mode), we will descend into TT as we move from left to right, depth-first in P. As a preview, let us consider what TT would look like after the entire plant, assuming TT then contained only this one pattern:



Display 32.

This structure arises as follows: The pattern, being complex and of length 3 at the top level (i.e. (KNOWS JOHN *)), comes to be stored in the complex field of the root node of TT, grouped with any other existing patterns of length 3. Descending into P, the first element of the top level list is

KNOWS, a constant. KNOWS therefore gives rise to a successor out of the second node of TT from within the constants field.

Moving across to the second element at the top level of P, another constant, JOHN, is encountered, giving rise to another node similar to the one for KNOWS. Moving to the third element of P and descending one node in the tree, a complex, (LOVES ...), is encountered, causing a successor from the third node of the trigger tree to be sprouted via the complexes field (again, subindexed by its length, 3). Descending, the first element of this nested n-tuple, LOVES, gives rise to another node with an edge leading out of its constants field.

Moving to the second and third elements of the nested list causes the remainder of the tree to be built up in an analogous manner. Thus, going from a node to its successor in TT corresponds to moving right one element in P, and descending into the new element in case it is complex.

A variable at position X in the pattern P being stored in TT will cause an arc to be created out of the variables field of the node in TT corresponding to position X in P. Since the trigger tree is serving to unify many patterns into one structure in which common initial paths are shared, it is necessary to maintain a uniform variable naming convention.

To ensure a uniform naming convention, as a pattern is stored, each of its variables is mapped into a variable of a canonical name that is derived from its position in the pattern being stored. A position is described by a <path>, as previously defined.

In our convention, if variable V occurs at position R in P, V will be known within the trigger tree as the variable whose print name is "-X" concatenated with the path denoting R.

For example, if the pattern to be stored in the tree is

(KNOWS -X (LOVES -Y -Z))

Display 33.

---

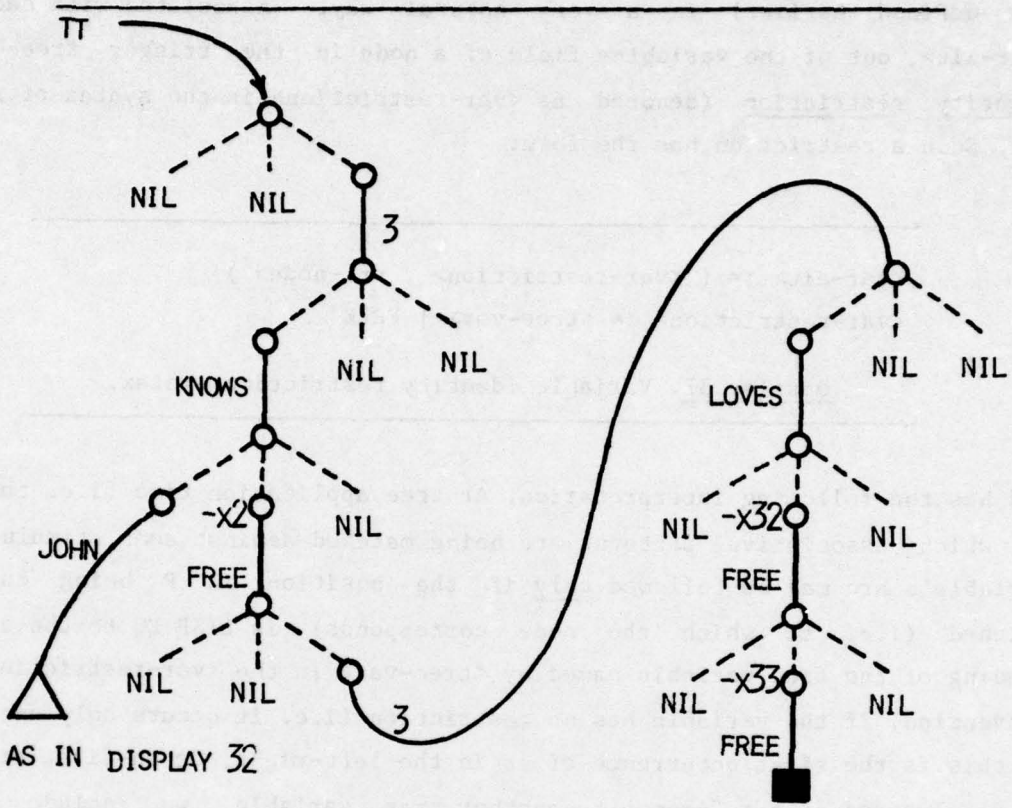
then within the tree the pattern will be known as:

(KNOWS -X2 (LOVES -X32 -X33))

Display 34. Canonical variable naming.

At the terminal node of the tree which corresponds to this pattern, variable mapping information of the form "-X2 is actually -X, -X32 is actually -Y, -X33 is actually -Z" will be stored so that when the pattern is associatively accessed by a tree traversal in response to some stimulus, original variable names may be reconstructed and tree variable bindings transferred to the variables as they are known to the SC of which the tree pattern is a part.

Suppose now that we were to store the pattern of Display 34 with three variables in the trigger tree we have begun to construct in Display 32. Then, the new, augmented tree would be:



Display 35. Trigger tree TT, now containing two associative patterns.

Now we have introduced some variables into TT. When the variables field of a trigger tree node first becomes non-NIL, the variables field of that node blossoms from NIL to the form:

(<tree-var> <var-alt> ... <var-alt>)

Display 36. Trigger tree node variable field syntax.

where <tree-var> is the variable of canonical name corresponding to the pattern position the node in the trigger tree represents. Storing the tree variable in explicit form will make it a simple matter to bind this variable during associative tree accesses.

Trigger trees will permit us to perform variable identity restrictions

(as defined earlier) in a very natural way. Associated with each arc, <var-alt>, out of the variables field of a node in the trigger tree is an identity restriction (denoted as <var-restriction> in the syntax of Display 28). Such a restriction has the form:

<var-alt> := (<var-restriction> . <tt-node>)
<var-restriction> := <tree-var> | FREE

Display 37. Variable identity restriction syntax.

---

and has the following interpretation. At tree application time (i.e. the time at which associative patterns are being matched against some stimulus), the variable's arc may be followed only if the position in P being currently matched (i.e. to which the node corresponds) is LISP EQ to the existing binding of the tree variable named by <tree-var> in the <var-restriction>. By convention, if the variable has no restriction (i.e. it occurs only once in P, or this is the first occurrence of it in the left-right, depth-first traversal on P), instead of a reference to another tree variable, we include as the <var-restriction> field of <var-alt> the distinguished atom FREE.

This syntax will permit us to discriminate triggers on the basis of variable identity in addition to the structural and constant information within the pattern. For example, if we now augment the trigger tree, TT, we have been building with a third, very similar pattern:

(KNOWS -X (LOVES -Y -X))

Display 38.

---

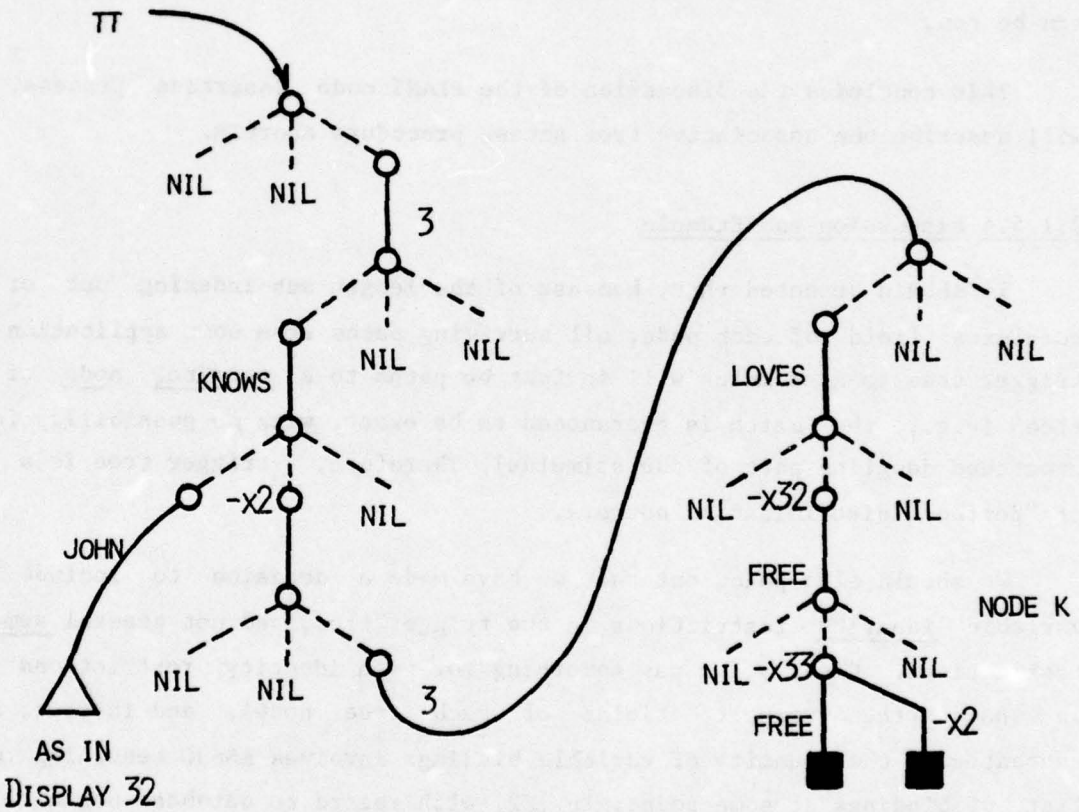
which becomes:

(KNOWS -X2 (LOVES -X32 -X33))

such that -X33 = -X2

Display 39.

within the tree, the new tree will be:



Display 40. Introducing some variable identity restrictions.

The variable field of the node marked K in this tree is read as follows: the arc leading from the restriction denoted by FREE can always be followed unconditionally at associate-time; additionally, the arc leading from the restriction denoted by -X2 can be followed only in the event that the current position, corresponding to tree variable -X33, is LISP EQ to the existing

binding for -X2 (which was established higher in the tree).

As a variables field arc is followed during an associative tree access, a record of the binding which had to be performed in order to follow the arc will be made, and, along with prior existing bindings, is passed to lower levels of the tree. At the end of a successful traversal to a terminal node, all tree variables encountered along the traversal path will have bindings. They can then be translated back into the original names for the various variables in the pattern the successful path represents, and then passed to the body of the SC in case the ensuing polling process determines that the SC can be run.

This concludes the discussion of the PLANT mode insertion process. We will describe the associative tree access procedure shortly.

2.1.5.4 Discussion and Example

It should be noted that, because of the length sub-indexing out of the complexes field of each node, all surviving paths from some application of a trigger tree to a stimulus will in fact be paths to a terminal node of the tree (e.g., the match is guaranteed to be exact, with no possibility for an unmatched dangling part of the stimulus). Therefore, a trigger tree is a kind of "perfect" discrimination network.

We should also point out that we have made a decision to include only variable identity restrictions in the trigger tree, and not general semantic restrictions. Clearly, we pay something for even identity restrictions both in space (the <var-alt> fields of each tree node), and in time, since guaranteeing the equality of variable bindings involves ASSOC searching on a list of bindings at some point. In [M2] with regard to database organization, McDermott has argued against making identity restrictions during the first stages of access, arguing that variable identity restrictions are more efficiently processed after the initial filtering on constant and structural bases. However, we feel that the potential pruning effect this has on the accessing of large trigger trees will be well worth the cost.

It should also be noted that it would be nearly trivial (syntactically) to include arbitrary semantic restrictions on tree variables simply by

allowing the <tree-var> part of each <var-restriction> to be an arbitrary S-expression which would have to evaluate non-NIL in order for the associated arc to be taken. However, here we would agree with McDermott that this type of variable restriction is better handled after the trigger tree has finished (e.g., as other components of the complex trigger pattern which are polled after the initial activation, as discussed earlier), since its path-pruning contribution would probably be negligible in relation to the amount of additional run time it would incur.

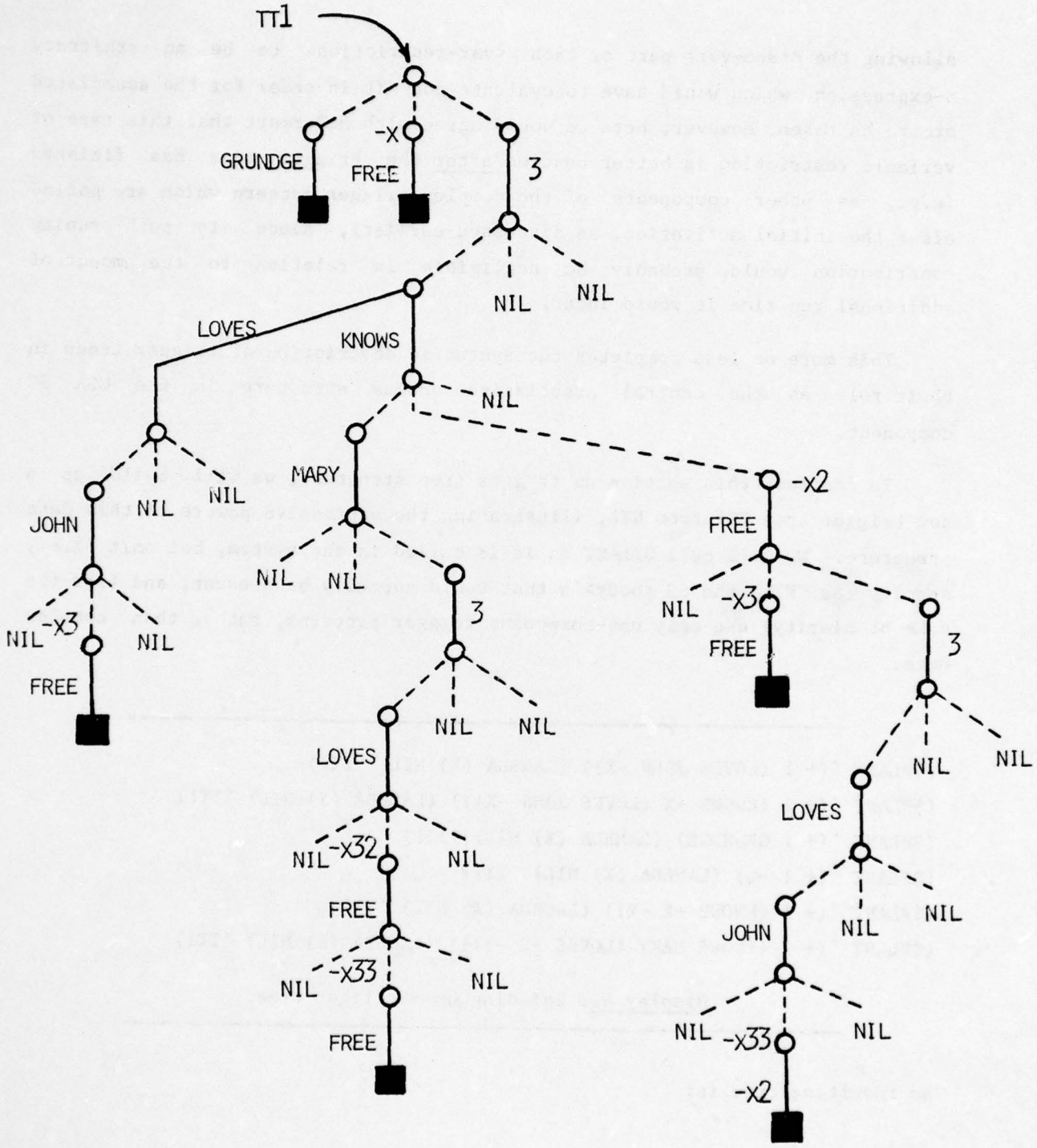
This more or less completes the syntactic description of trigger trees in their role as the central associative access structure in the CSA SC component.

To conclude this section on trigger tree structure, we will build up a new trigger tree TT1 from NIL, illustrating the expressive powers of this data structure. We will call $PLANT as it is called in the system, but omit (i.e., specify as NIL) the SC <body>'s that would normally be present, and (for the sake of clarity) use only one-component trigger patterns, rather than complex ones.

```
($PLANT '(+ 1 (LOVES JOHN -X)) (LAMBDA (X) NIL) 'TT1)
($PLANT '(+ 1 (KNOWS -X (LOVES JOHN -X))) (LAMBDA (X) NIL) 'TT1)
($PLANT '(+ 1 GRUNDGE) (LAMBDA (X) NIL) 'TT1)
($PLANT '(+ 1 -Q) (LAMBDA (X) NIL) 'TT1)
($PLANT '(+ 1 (KNOWS -X -Y)) (LAMBDA (X) NIL) 'TT1)
($PLANT '(+ 1 (KNOWS MARY (LOVES -X -Y))) (LAMBDA (X) NIL) 'TT1)
```

Display 41. Building up a trigger tree.

The resulting tree is:



Display 42. Resulting tree.

and, by calling the function ($SHOW <trigger-tree>) (Appendix A), we can see TT1's LISP manifestation in a formatted form:

($SHOW TT1):

```
                    GRUNDGE
                       (G3 NIL NIL)
                    (-X = FREE)
                       (G4 NIL (-Q))
                    (3)
                       KNOWS
                          MARY
                             (3)
                                LOVES
                                   (-X32 = FREE)
                                      (-X33 = FREE)
                                         (G6 NIL (-Y -X))
                          (-X2 = FREE)
                             (-X3 = FREE)
                                (G5 NIL (-Y -X))
                             (3)
                                LOVES
                                   JOHN
                                      (-X33 = -X2)
                                         (G2 NIL (-X))
                    LOVES
                       JOHN
                          (-X3 = FREE)
                             (G1 NIL (-X))
```

Display 43. LISP form of the tree.

2.1.6 Trigger Tree Terminal Nodes

A path in a trigger tree corresponds to one associative part of a complex

trigger. (Of course, complex triggers which reference the same associative component will share paths in a trigger tree). Beyond the associative information inherent in the path to a terminal node, each terminal node of a trigger tree must contain information about what to do in case some stimulus causes that terminal node to be reached during an associative access.

Specifically, we will require the following information at each terminal node in a trigger tree:

(1) a distinctive marking to distinguish the terminal node from a non-terminal node

and for each spontaneous computation, S, whose complex trigger pattern includes a reference to the associative component represented by this terminal node,

(2) a reference to the entity in the system (a LISP GENSYM in our case) which represents S itself

(3) the polling list, L(A), of other parts of S's complex trigger pattern components (produced by $FRAGMENT) which must be polled and found to be true before S can be invoked, and

(4) a list of variable mappings which will translate the bindings attached to tree variables to the variables as they are named in the original declaration of S's complex trigger pattern.

We thus define a trigger tree terminal, <tt-t-node>, as follows:

<tt-t-node> := (<parent> SC <sc-ref> ... <sc-ref>)
<sc-ref> := (<sc> <polling-list> <var-map>)
<sc> := a LISP atom with CSA type "SC"
<polling-list> := NIL | (<tp> ... <tp>)
<var-map> := NIL | (<variable> ... <variable>)

Display 44. Trigger tree terminal node syntax.

---

where <tp> is a complex trigger pattern, as defined in Display 15.

To illustrate the form of a trigger tree terminal node, suppose we plant the following complex trigger pattern in an initially null trigger tree TT2, calling $PLANT:

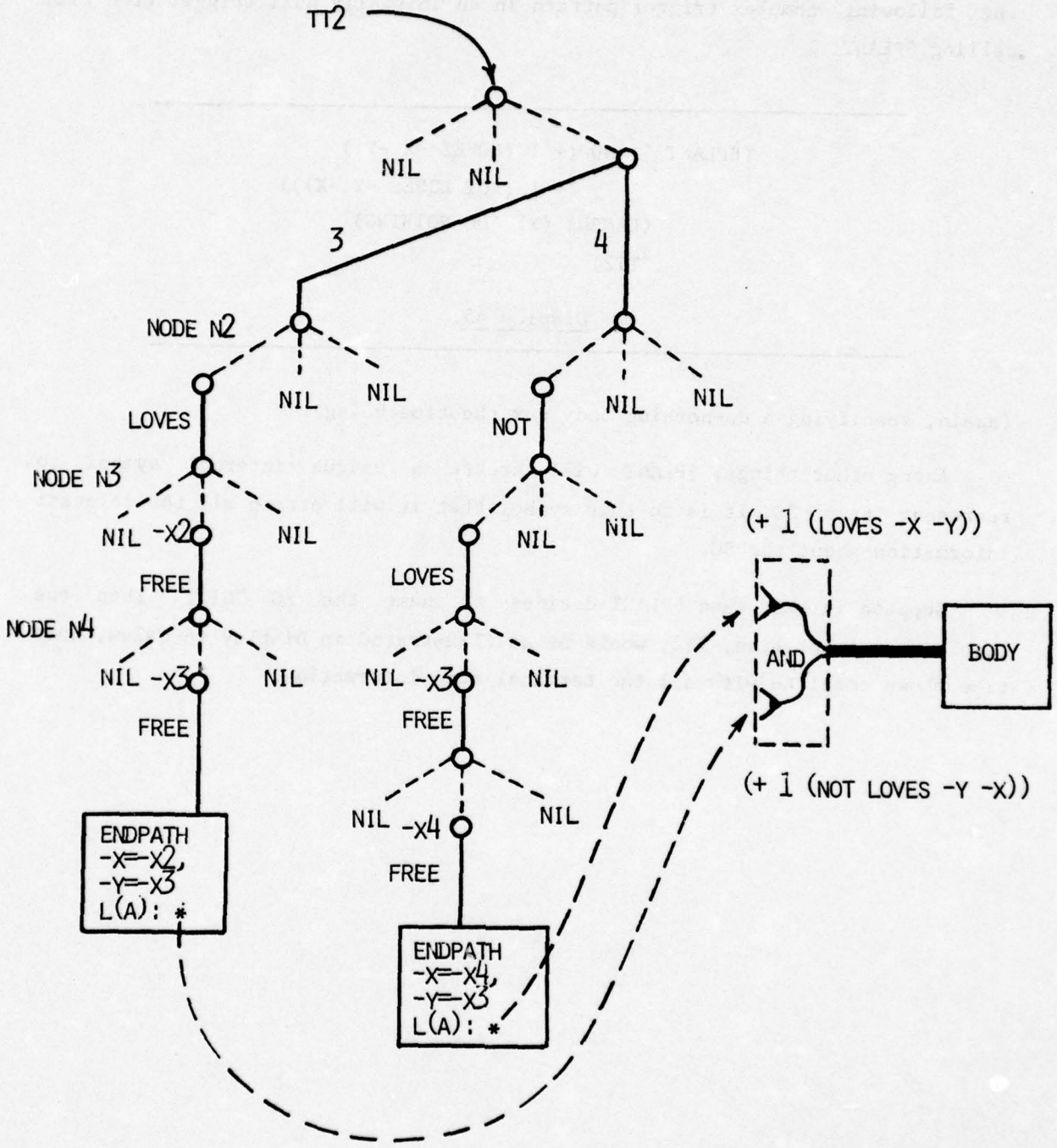
```
($PLANT '(AND (+ 1 (LOVES -X -Y))
              (+ 1 (NOT LOVES -Y -X)))
        (LAMBDA (X) 'DO-NOTHING)
        'TT2)
```

Display 45.

(again, specifying a do-nothing body for the time being).

Among other things, $PLANT will create a unique internal symbol to represent this SC. It is to this symbol that it will attach all the relevant information about the SC.

Suppose in this case $PLANT decides to name the SC "G1". Then the resulting trigger tree, TT2, would be as illustrated in Display 46 below, this time shown complete with all the terminal node information.



Display 46. Example trigger tree terminal node.

2.1.7 The Structure of an SC

The SC itself which comes to be created by $PLANT is normally a LISP GENSYM whose property list contains all the relevant information about the SC. This information is:

SC-PATTERN the complete complex trigger pattern exactly as it was communicated via the call to $PLANT

SC-BODY the LAMBDA expression of one argument which is the SC's body; the single argument will receive a list of dotted pairs representing the bindings which have caused the SC to be invoked

TREE-LOCS backpointers to a set of trigger tree terminal nodes which represent the associative components of the SC; these will be required in case the SC needs to be deleted

In addition to these properties, there is a property related to the context status of the SC, and some properties related to the manner in which the SC is to be handled at invocation time. The latter will be discussed later; we will not describe the CSA context mechanism (a fairly straightforward chronological context scheme) in this paper, other than to include the code as Appendix D.

2.1.8 SC Associative Tree Access and Invocation

We are now ready to describe the flow of control during an associative trigger tree access. A trigger tree is caused to react to a stimulus by calling the function $ACTIVATE:

($ACTIVATE <stimulus> <trigger-tree>)

Display 47. Causing an associative access.

For instance, if we want TT2 (as defined in Display 46 above) to react to the stimulus (LOVES JOHN MARY), we would call:

($ACTIVATE '(LOVES JOHN MARY) 'TT2)

Display 48.

Alternatively, it is possible simply to call:

(TT2 '(LOVES JOHN MARY))

Display 49.

---

since, as any trigger tree is created it is simultaneously defined as a LISP function (an EXPR) which can "apply itself" to a given stimulus. This construction was motivated by some considerations about trigger tree control which we will come to later.

$ACTIVATE is first concerned with locating all SCs which nibble at the stimulus pattern. To do so, $ACTIVATE calls another function, $NIBBLERS, whose two arguments are the stimulus and the tree. The sequence of events that occurs in response to this call on $NIBBLERS is as follows: The root node of the trigger tree, in this case TT2, is accessed, and a test posed: is the "current object" (initially the entire stimulus, (LOVES JOHN MARY)) a constant (i.e. LISP atom) or a complex object? In this case, it is a complex, so the complex field of the root node of TT2 is searched for an arc out representing complexes of length 3. One is found, namely the node marked N2 in Display 46; hence, $NIBBLERS descends into the current object, preparing to move left to right, making the new current object LOVES. Additionally, $NIBBLERS asks whether there are any variables in the variables field of the root node which could bind to the initial current object. In this case, there are no variables at all, so no paths begin from the variables field. If some were begun, they would be pursued "in parallel" with the path we will pursue here, until they either died out or resulted in successful matches.

At N2, having been recursively entered, $NIBBLERS poses the same questions: is the current object a constant or a complex? This time, it is the

constant LOVES, so the constants field of N2 is consulted to determine whether there is an arc out associated with LOVES; finding one, $NIBBLERS moves to N3 in TT2, simultaneously making the new current object the next one to the right of LOVES in the pattern, namely JOHN. As before, at N2 $NIBBLERS also tries to match the current object to any relevant variables, but, finding none, again pursues only the constant LOVES path.

At N3, the process repeats: is JOHN a constant or a complex? Since it is constant, but there is no arc out of N3 associated with the constant JOHN, no path is followed out of the constants field. But this time there is a FREE variable to which to bind JOHN, namely -X2. The binding is made, and $NIBBLERS follows the associated arc to N4, passing the binding just made down to lower levels of this path. The new current object then becomes MARY.

At N4 an event similar to the event at N3 occurs, namely, -X3 is bound to MARY. Then TT2's single terminal node, N6 is encountered. Because we have implicitly taken nested complexes' lengths into consideration in the paths $NIBBLERS has followed, we know that $NIBBLERS has a non-dangling match, and furthermore, that the bindings are:

((-X3 . MARY) (-X2 . JOHN))

Display 50.

(and in fact always in order... the order will turn out to be useful).

$NIBBLERS thus returns with a list of trigger tree terminal nodes representing successful associative paths. With each terminal is associated the list of bindings that were made along the path.

Next, for each SC reference on the terminal node's list of SCs of whose complex trigger pattern this stimulus is a part, $NIBBLERS

(1) associates with each complex trigger pattern variable the relevant binding, after mapping tree variables into the variables as they were named in the pattern; in this case, this results in: ((-X . JOHN) (-Y . MARY))

(2) accesses the polling list (associated as L(A) with A= (LOVES -X -Y)) of remaining components of the complex trigger pattern for this SC, and instantiates this list, substituting the variables -X and -Y with their associated constant bindings (i.e. creating an instantiated copy). It should be noted that in general (but not in this case), the instantiated result will still contain variables which were part of the complex pattern, but not mentioned in the associative component which has given rise to the polling;

(3) polls the resulting patterns to determine whether the SC is ready to run.

In the case of this example, the polling will attempt to determine whether or not it is true that (NOT LOVES MARY JOHN) via a call that looks like:

(+ 1 (NOT LOVES MARY JOHN))

Display 51.

"True" will mean: can the database/deductive components of the system decide on a truth value for this fact, within the allowed energy budget specified in the <energy> field of this call as it was specified in the complex trigger pattern. In this case, since we have limited the budget to exactly one fetch, the SC will be run if and only if (NOT LOVES MARY JOHN) is explicitly in the database.

This interaction between association on the one hand via trigger trees, and polling on the other hand via the deductive components of the system will provide a basis for some intriguing theoretical interactions; we will discuss this more in the second half.

2.1.8.1 Polling and $ALLBINDS

The code which, given some component of the complex trigger to be polled,

carries out the polling (search for ways to instantiate the component's variables in order to make it true), is called $ALLBINDS. This function is basically no more than a control and accumulating function which drives the deductive component in ways specified by the pattern.

$ALLBINDS will accept any complex trigger pattern, <tp>, as defined earlier, and control a sequence of calls on the database/deductive components of the system, accumulating a list of ways the pattern can be caused to become true (or, in the case of a pattern with no variables, compute the truth of the pattern).

As such, $ALLBINDS is of general theoretical utility for converting opaque references (references by features) called descriptors in the CSA system, into identity references (pointers to internal memory tokens and concepts within the CSA system). Given a complex descriptor, $ALLBINDS will return either NIL, in case there is no way to satisfy the complex pattern, or the form:

(T <binding-list> ... <binding-list>)

Display 52. $ALLBINDS response form.

$ALLBINDS is scheduled to be developed more in the near future, so we will defer any further discussion of it until a later paper. The $ALLBINDS code, rudimentary as it is, is included in Appendix C.

2.1.8.2 The SC Body and Invocation Control

Returning to our (LOVES JOHN MARY) activation example, assuming that a successful polling has occurred, the SC's body is ready to be invoked. In our example, only one copy of it will be invoked. However, if there had been any variables left in the components of the complex trigger which were polled, it could have happened that, within the cumulative energy budgets, there were numerous ways to bind the remaining variables. For example, if the original trigger pattern had been:

```
(AND (+ 1 (LOVES -X -Y))
     (+ 1 (NOT LOVES -Y -X))
     (+ 1 (LOVES -Y -Z)))
```

Display 53. A more combinatorial pattern.

$ALLBINDS would have seen two items for polling, instantiated as follows for the associative stimulus (LOVES JOHN MARY):

```
(+ 1 (NOT LOVES MARY JOHN))
and (+ 1 (LOVES MARY -Z))
```

Display 54. Pattern to be polled.

If Mary can be found to love more than one person, say PETE and JACK, we will now have two ways to instantiate the -X, -Y, -Z of this pattern:

```
(1)  ( (-X.JOHN) (-Y.MARY) (-Z.PETE) )
(2)  ( (-X.JOHN) (-Y.MARY) (-Z.JACK) )
```

Display 55. Multiple invocations from one stimulus.

and are confronted with the problem of making an interpretation of this situation.

We choose the obvious one: invoke two copies of the SC, calling one with the first alternative instantiation, the other with the second. This seems to be the only reasonable interpretation, since the ways to bind variables during polling can become rather involved.

The actual invocation of an SC whose trigger pattern has survived polling happens as follows: An SC body is always defined by a LAMBDA expression of the form:

```
(LAMBDA (X) <s1> ... <sn>)
```

Display 56. Form of an SC body.

where <s1>,...,<sn> carry out the SC's computation. As each SC is created, it, like a trigger tree, is simultaneously defined as a function whose LAMBDA expression is as shown above. This means that if, say, some SC is known internally as G1, invoking it will amount simply to calling it with its single argument bound to the list of bindings being passed in from the trigger pattern:

```
(G1 <binding list>)
```

Display 57. Invoking SC G1.

where a <binding list> is a list of dotted pairs associating variables and constants.

In the CSA system, $ACTIVATE does not actually cause the invoked SCs to be run; rather, it constructs a queue of calls such as these and returns the queue as its response (or, in case it is called with an optional third argument, $ACTIVATE will augment an existing queue named by the optional argument). As it queues each SC call, it will associate a run condition with the call. The interpretation of an SC run condition is: even though the SC has been successfully queued, to run, it will not actually be allowed to run until the run condition (an EVALable LISP expression) becomes true. We will discuss this more later on.

2.1.9 SCs and Context

Every SC in every trigger tree has an associated context (environment in which it is to be considered active). SCs can therefore be masked and unmasked in various context levels by the general CSA context functions $HIDE and $UNHIDE. The code for these is included in Appendix D. In fact, as will be

described shortly, entire trees can be masked and unmasked, so that we have the possiblity for one SC annihilating another SC, or controlling an entire population, etc.

We have now more or less reached the end of our treatment of the engineering issues of how to store and access spontaneous computation complex trigger patterns. We are ready now to consider how to harness this type of computation in a higher level control paradigm.

2.2 Higher Level Control of Spontaneous Computation

The question of what to do with SCs when they are ready to run is very much related to what it was that caused them to want to run. Therefore, let us return now to the metaphor of the central arena which some population of SCs has been watching, and ask: what types of activities are reasonable to monitor?

In the two languages which have employed SC as a central paradigm, PLANNER and CONNIVER, the activities to which SCs react are limited to two specific types:

(1) changes to a central database

(2) requests for service on a "hot line" to which many computational modules have access.

"Database change" means either the storage or erasure of an S-expression from the database; in PLANNER, the SCs which monitor stores and erasures are, respectively, THANTE and THERASING "theorems", while in CONNIVER, they are called IF-ADDED and IF-REMOVED "methods". An SC which monitors the hot line for a request which matches its (simple) trigger pattern is called a THCONSE theorem in PLANNER, an IF-NEEDED method in CONNIVER. Thus, the central arenas in these languages are limited to database changes and the hot line.

The approach to SC trigger pattern organization we have developed makes it natural to group SCs into populations in the sense that each trigger tree could, e.g., be thought of as a functional group of watchers tuned to some specific part of the environment, some specific phase of an operation, or some specific context. There may then either be one large, system-wide population,

as there is in PLANNER and CONNIVER, or there may be numerous small trees. In the latter arrangement, trees would perhaps pay exclusive attention to one arena, or perhaps each tree would gather its perceptions from several arenas.

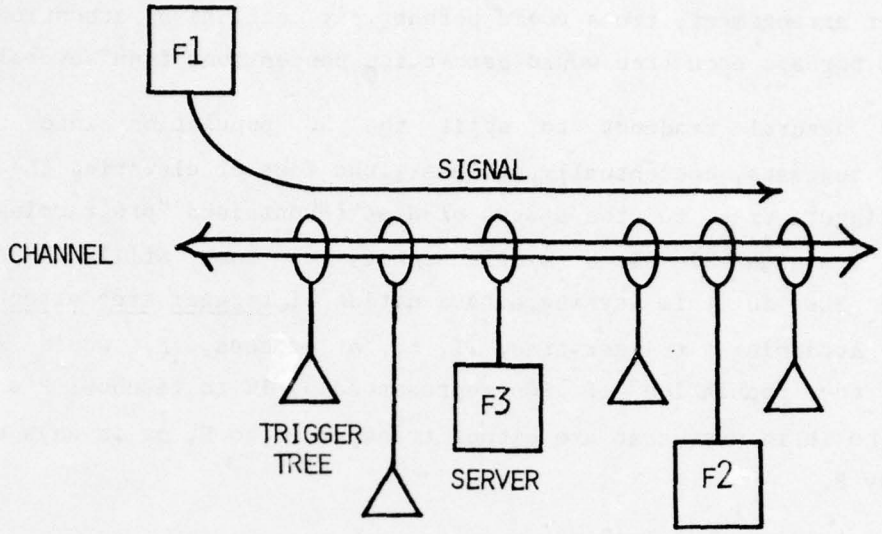
This natural tendency to split the SC population into functional subgroups suggests, conceptually at least, the idea of elevating the notion of a SC trigger tree to the status of a self-contained "programming language construct", manipulable as a single entity at some still higher level. Regarding SCs in this way suggests a notion of trigger tree attachment to a process. Attaching a trigger tree, TT, to a process, P, would amount to allowing the population of SCs represented by TT to fishbowl P's activity, reacting to it in ways that are either transparent to P, or in ways that alter or destroy P.

2.2.1 Channels

If this is our vision of SC populations, to what are we to attach trigger trees? In PLANNER and CONNIVER, there is one large population attached to the store function, one large population attached to the erase function, and one large population more or less hard-wired into the central control for the system, i.e. the hot line. From these ideas, we have made the (intellectually modest) leap to the notion of a channel.

We define a channel to be the medium whereby one LISP function calls (posts requests to) another LISP function. Doing this will essentially allow us to "make public" what is ordinarily the private calling protocol between functions, the locus of all the real work in LISP. This metaphor of a channel will also give rise to a new programming construct, the CHANNEL (!), to which trigger trees may now be attached.

We will visualize a channel as follows:



Display 58. Graphic representation of a channel.

Now, wherever function F1 used to call function F2 via the standard LISP protocol, we will now require F1 to post all requests to F2 on this intermediate construct, the channel. Unbeknownst to either F1 or F2, we will now admit the possibility of one or more trigger trees, as well as other functions, such as F3 in Display 58, being attached to this channel, either as "transparent", benign watchers, or as "modifying", possibly inimical watchers.

Since all the work in a LISP environment transpires via function calls, if we give watchers the ability to see calling sequences, we will have a "most general" SC attachment paradigm. Although we will tend to regard the new channel construct as a state of mind (i.e. a convention not enforced by LISP), one could imagine enforcing this style of communication by restructuring LISP's control. Doing so would bring us into a realm of thinking akin to Hewitt's "actors" and "messages" paradigms [H2]. (In fact, in retrospect, the whole concept of channels and trigger tree attachment fits in very nicely with Hewitt's view of computation.)

We are about to present a specific model of channels and trigger tree attachment which develops the notions via a hardware channel metaphor. But before we do, we ought to point out that the new notions will apply to a paradigm in which SCs can react only to changes in states of computation,

rather than to states of computation themselves. Ordinarily, this will not be a severe restriction, since most reasonable systems will be built up or taken apart incrementally, and incremental model evolution fits our SC paradigm precisely. However, it will not be possible, say, to confront an SC system such as we are developing with a snapshot of a database and say "react"! This is another style of spontaneous computation, which, although it could be simulated by methodically lifting pieces out and putting them back in again while populations of our SCs were watching, seems to be fundamentally different; it is in some sense "far more associative", since it does not rely on triggers, but rather on configurations. We will not pursue this style on SC in this paper, but will keep it in mind as a future idea. **

** We are always interested in staying close to our intuition about how the human brain must work. Even if we could engineer some exotic theory and/or hardware to react to configurations, it is not clear that this would be a good model of human associative powers anyway; the feeling is that the model we have been developing coincides with some key intuitions about how our brains work.

2.2.2 Channel Characteristics

A channel is a construction with the following features:

(1) it has a one-dimensional "extent", with directionality

(2) other constructs can be attached to it at tap points; there is no limit to the number of tap points

(3) the left-right ordering of tap points is significant

(4) each tap point is either a watcher or a server, and have mode either transparent or modifying

(5) signals (either requests to a server, or a response from a server) may be injected on a channel at arbitrary starting points, and propagate either left or right.

(6) both channels and tap points are context-sensitive, so that reconfiguring the medium by which one function calls another is relatively simple.

In this manner, any given trigger tree may be attached to any number of channels at any number of points.

The primary channel-related functions in the current CSA system are: **

```
($CONNECT <object> <channel> <mode> <type>
          <in-relation-to> <other-point>)

($DISCONNECT <object> <channel>)

($INJECT <signal> <server> <channel>
         <in-relation-to> <other-point> <prop-direction>)
```

Display 59. The 3 primary channel functions.

** The code for all channel-related functions is included as Appendix B.

where the arguments to these functions are defined as follows:

```
<object> := <watcher> | <server>
<watcher> := <trigger tree name> | <LISP function name>
<server> := <LISP function name>
<channel> := <LISP atom>
<mode> := TRANSPARENT | MODIFYING
<type> := WATCHER | SERVER | RESPONSE-WATCHER
<in-relation-to> := BEFORE | AFTER | AT
<other-point> := <watcher> | <server> |
                 RIGHT-END | LEFT-END
<signal> := <LISP S-expression>
<prop-direction> := LEFT | RIGHT
```

Display 60. Channel functions argument syntax.

and the semantics of <other point> are that it be some existing tap point on the channel.

2.2.3 Channel Operation

Since a channel is both spatial and directional, we will imagine a signal to propagate from some starting point in some direction with finite speed. As it passes by a tree of watchers, any relevant watchers in the tree will be triggered and run, and either (1) allow the signal to continue as-is, (2) modify the signal but allow it to proceed, or (3) block the signal altogether. If and when the signal reaches the requested server, the server will be run unconditionally on the (possibly modified) signal. Its response will then be momentarily held while the signal is allowed to propagate to the end of the channel, or until it is blocked. At that time, the server's response will be injected on the channel, starting at the server's tap point. The response is defined simply as the LISP value the server returns, and it will propagate from the server's tap point of the original request that instigated the channel activity. On its way back, the response may pass over a set of "response watchers" which, similarly, can have the potential for altering or blocking the response as it passes on its way back to the requestor.

As an illustration, we will set up a rather simple channel configuration which models the THANTE capability in PLANNER. This will amount to activating any relevant watchers after some fact has successfully entered the system's database.

The CSA calls required to set up this channel are:

```
($CONNECT '$STORE 'DB-IN-CH 'TRANSPARENT 'SERVER
          'AT 'RIGHT-END)

($CONNECT 'TRIGGERTREE1 'DB-IN-CH 'TRANSPARENT 'WATCHER
          'AFTER '$STORE)
```

Display 61. Setting up a channel to model PLANNER.

i.e., attach the function $STORE (the CSA database storing function) to channel DB-IN-CH (creating this channel if it does not already exist) as a transparent server at the right end of the channel; then attach TRIGGERTREE1 to DB-IN-CH as a transparent watcher after (to the right of) $STORE.

Now, wherever $STORE would have been called directly, as in

```
($STORE '(LOVES JOHN MARY))
```

Display 62. The old way to store facts.

---

we will in the future store facts by placing them as signals to $STORE on the DB-IN-CH:

```
($INJECT '(LOVES JOHN MARY) '$STORE 'DB-IN-CH
         'AT 'LEFT-END 'RIGHT)
```

Display 63. The new way to store facts.

---

that is, inject the signal (LOVES JOHN MARY) to the $STORE server on channel DB-IN-CH, starting at the left end of the channel, propagating right.

If, on the other hand, we wished the population of SCs in TRIGGERTREE1 to have access to the signal before $STORE sees it, and furthermore, to be able to modify or block the signal altogether, we would set up DB-IN-CH as:

```
($CONNECT '$STORE 'DB-IN-CH 'TRANSPARENT 'SERVER
          'AT 'RIGHT-END)

($CONNECT 'TRIGGERTREE1 'DB-IN-CH 'MODIFYING 'WATCHER
          'BEFORE '$STORE)
```

Display 64.

---

and then inject requests from the left end.

Of course, by knowing the structure of the channel, we could sneak signals past this population of modifying watchers either by temporarily disconnecting them:

```
(SDISCONNECT 'TRIGGERTREE1 'DB-IN-CH)

(SINJECT '(LOVES JOHN MARY) 'SSTORE 'DB-IN-CH
         'AT 'LEFT-END 'RIGHT)

(SCONNECT 'TRIGGERTREE1 'DB-IN-CH)
```

Display 65. A temporary disconnect-reconnect sequence.

or, more simply, by injecting the signal to $STORE at a point where TRIGGERTREE1 would be bypassed, or seen only after $STORE had seen the signal:

```
(SINJECT '(LOVES JOHN MARY) 'SSTORE 'DB-IN-CH
         'AFTER 'TRIGGERTREE1 'RIGHT)

or

(SINJECT '(LOVES JOHN MARY) 'SSTORE 'DB-IN-CH
         'AT 'RIGHT-END 'LEFT)
```

Display 66. Putting the directionality to use.

If, additionally, we desired that a server's response be monitored, we would attach a trigger tree of response watchers, either blocking or transparent, at some point on the channel where response signals from the server back to the requestor were expected to pass. For example, in the CSA system, $STORE will return a GENSYMmed atom as its response, indicating where the new fact has been stored. Knowing this, our "population" of $STORE response watchers need be nothing more than a single SC whose trigger pattern is simply:

```
(+ 1 -X)
```

Display 67. A simple response watcher.

(i.e. a single variable to which the passing GENSYM would bind). The body of this single SC could then manipulate or change the new fact in any way required, and possibly modify the signal as it propagated back to the injector.

We would stage such a situation by:

```
($PLANT '(+ 1 -X) <some body> 'TRIGGERTREE2)

($CONNECT 'TRIGGERTREE2 'DB-IN-CH 'MODIFYING
          'RESPONSE-WATCHER 'ATFER 'LEFT-END)
```

Display 68. Setting up a response watcher for $STORE.

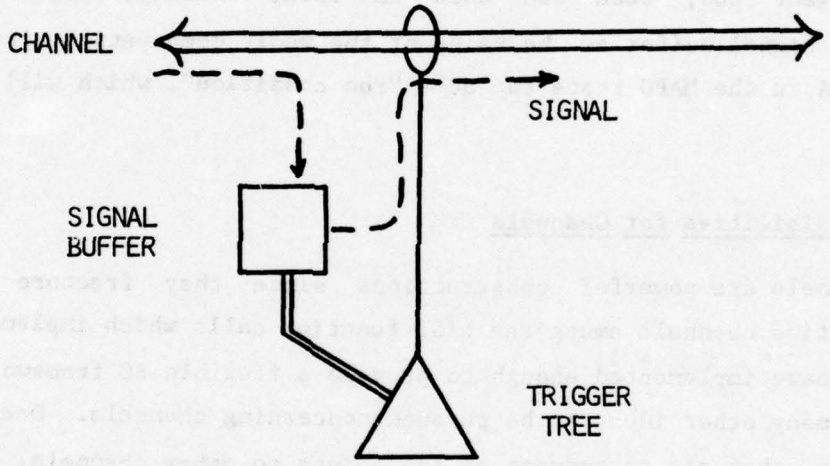
2.2.4 Tap Points

A watcher need not be a trigger tree; instead, it is allowed to be an arbitrary EXPR of one argument. This is in reality how a trigger tree looks to a channel anyway. In the implementation, as a signal passes by a watcher, the watcher is simply APPLYed to the signal. If the watcher is a modifying watcher, W, the signal which will propagate past W is simply the value returned by W.

In order to make trigger trees compatible with this protocol, every trigger tree (as it is created) receives a function definition as well as the tree structure itself. As shown earlier, this allows us to regard the tree as an EXPR of one argument which can be "applied to a stimulus" in order to activate some subset of its population of SCs. Hence, as a signal passes by a trigger tree, T, T is simply APPLYed to the signal.

In the case of a modifying watcher, W, which is a function (rather than a trigger tree), the modified signal to be propagated can be defined simply as the value W returns. But for trigger trees, which are mobs of computations not necessarily related to one-another or coordinated in any way, and where there could be quite a few relevant SCs which run in reaction to some signal, there is a question of how the modified signal is to be computed and communicated back to the channel.

To solve this dilemma, each trigger tree has an associated signal buffer, depicted as:



Display 69. Trigger tree signal buffer.

As the tree is about to be applied, its signal buffer is initialized to the original signal. Any SC that wishes to alter the signal does so simply by replacing the contents of its tree's signal buffer with some new value. The value in the buffer after all SCs have been run is the signal to be propagated.

By definition, when a watcher modifies the signal to NIL, the signal has been blocked, and its propagation down the channel ceases.

The LAMBDA expression which defines a trigger tree as a function as it is created has the following form:

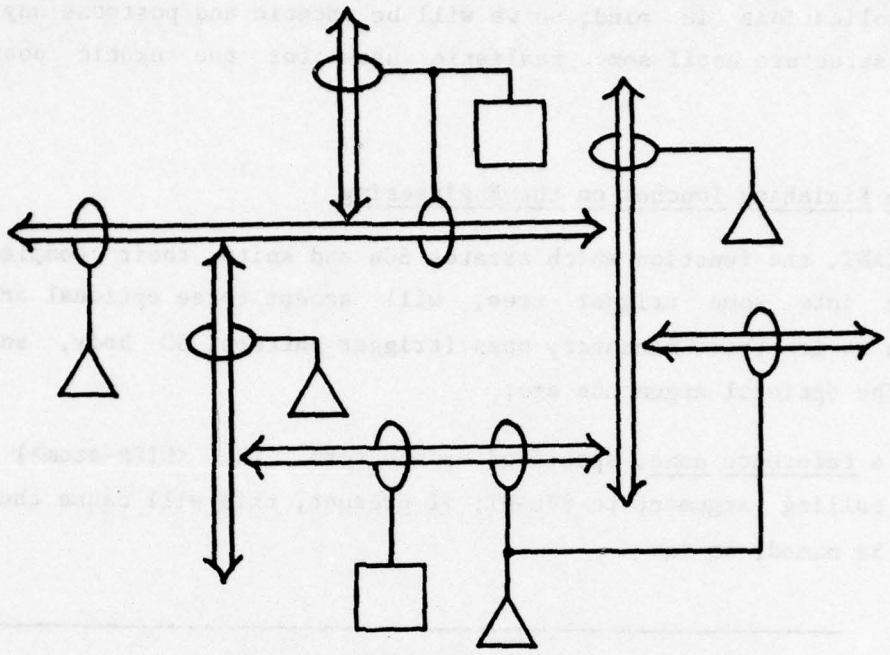
```
(LAMBDA (STIMULUS)
  (PUT <tree name> 'RESPONSE-BUFFER STIMULUS)
  (MAPC ($ACTIVATE STIMULUS <tree name>)
        (LAMBDA (SC) (COND ((EVAL (CDR SC))
                            (EVAL (CAR SC))))))
  (GET <tree name> 'RESPONSE-BUFFER))
```

Display 70. Trigger tree functional definition.

i.e., for a given STIMULUS, initialize the buffer to this stimulus, $ACTIVATE all relevant SCs, then run each in turn; finally, return the (possibly modified) signal buffer as the value of the whole computation. The COND inside the LAMBDA in the MAPC tests the SC's "run condition", which will be described shortly.

2.2.5 Possibilities for Channels

Channels are powerful constructions since they fracture the private communication channels among the LISP function calls which implement a theory. While we have implemented enough to provide a flexible SC framework, there are clearly many other ideas to be pursued concerning channels. One obvious idea is to allow channels to connect as tap points to other channels, giving rise to situations such as:



Display 71. Channel-channel interconnections.

Another obvious generalization would be to move away from requests to servers by name, and toward a more pattern-directed scheme wherein requests would be injected to (possibly multiple) servers on the basis of pattern matching the request pattern against the patterns advertizing the servers' capabilities. Of course, we already have this ability if, instead of distinguishing servers from watchers, we regard all servers as a special subclass of watchers. For example, instead of attaching $STORE to a channel directly as a server, we could fabricate an SC which would react to patterns of the form ($STORE -X), place it as the sole SC in some tree, then attach the tree to the channel as a watcher. $INJECT's would then look like

```
($INJECT '($STORE '(LOVES JOHN MARY)) 'DB-IN-CH
'AT 'LEFT-END 'RIGHT)
```

Display 72. Calling $STORE as a watcher.

We want to avoid the temptation of inventing control structures with no good applications in mind; so we will be ascetic and postpone any orgies in control structure until some realistic uses for the exotic possibilities arise!

2.3 Some Finishing Touches on the Engineering

$PLANT, the function which creates SCs and knits their complex trigger patterns into some trigger tree, will accept three optional arguments in addition to its three mandatory ones (trigger pattern, SC body, and trigger tree). The optional arguments are:

(1) a reference name, specified by the form '(N. <LISP-atom>) as a calling argument to $PLANT; if present, this will cause the SC to be named, as in:

```
($PLANT '(AND (+ 1 (UNSUPPORTED -X))
              (- 10 (DISTANCE -X EARTH ORDERMILES)))
        <some body>
        'TRIGGERTREE3
        '(N.EARTHGRAVITY))
```

Display 73. Naming commonsense earth gravity.

(2) a run-queue priority, currently either FRONT or REAR, specified by the form '(P.{FRONT,REAR}). This will tell $ACTIVATE what to do with the SC when queuing it up for subsequent running. $ACTIVATE will ordinarily form a list of SCs to be run, and simply return this queue as its value. However, $ACTIVATE may be called with an optional third argument which names some existing queue to which it should send SCs as it invokes them. In either case, each SC, via this priority property, may specify the manner in which the SC is to be queued: either placed at the FRONT of the queue (the default), or at the REAR. We expect eventually to upgrade this facility to accomodate numerical priorities computed by some more

sophisticated criteria; but for the time being, we have no need for this.

(3) a run condition, denoted by the form '(R.<condition-builder>) where <condition-builder> is an S-expression which will be EVALed immediately prior to queuing an activated SC for running. The form which <condition-builder> constructs will be associated with the SC invocation call on the run queue; the SC will not be permitted to run until this condition is true. For example, if we did not want some SC to run, even though activated, until some timer, TIMER, had ticked, say, 10 times, we would include a run condition with the SC as it was planted: **

```
($PLANT <some trigger> <some body> <some tree>
        '(R . (LIST 'GREATERP 'TIMER (PLUS TIMER 10))))
```

Display 74. Attaching a run condition to an SC.

An SC in a trigger tree which is attached to a channel will be run after a successful invocation only if its run condition evaluates non-NIL at the time the tree is activated by a passing channel signal; SCs whose run conditions are not satisfied at that time are simply discarded. Because of this, we ordinarily omit the run condition when defining an SC which will participate in a tree attached to a channel, letting the condition default to TRUE.

** Grinberg does this type of thing in the CSA Mechanism Simulator, described in [RG1].

Many extensions to this rudimentary SC invocation queuing scheme will be possible, and probably desirable. For example, in addition to run conditions, it will probably be useful to have abort conditions which would, upon becoming true, cause the SC to be purged from the run queue. Again, we have held back in this area until a better picture of what will be needed emerges.

We have now described most of the design and engineering points of the existing SC component of the CSA system. Appendices A through E contain, respectively, the code for the SC component, the channels component, the database component, the context component, and some miscellaneous related functions. The contents of these appendices form a closed system which may be coded in any LISP; to expedite this, we have included as Appendix F a list of known differences, and their ameliorators, between our version of Wisconsin LISP for the UNIVAC 1110 and other more widespread LISPs such as Stanford (UCI) LISP, MACLISP and INTERLISP.

3. THEORY OF SPONTANEOUS COMPUTATION IN COGNITIVE MODELS

At this point, we are ready to move into some more theoretical considerations about how SC can be put to use in cognitive models. We will begin with the interesting question: What is the relationship between the SC component and the deductive component of the system as manifest during the polling process? In particular, what effect ought partially triggered SCs have on the system?

3.1 Partially Triggered SCs

Each SC trigger pattern may be regarded as a spiny urchin (sophomoric metaphors notwithstanding!); when any of its spines are touched associatively by some passing object, the rest of the spines are set in motion, by a polling process. (Is this the way a real urchin works?) While the triggering is purely associative, the polling process which ensues is very deductive, or goal-directed in the sense that in its subsequent behavior, the system becomes (at least momentarily) motivated to seek out the polled conditions to enable the SC to run. In a very important way, therefore, SCs will comprise a basic source of goal direction.

One interesting question is: what ought to happen when some trigger fires associatively on some stimulus, causing polling to occur, but then not all the required components of the trigger pattern can be derived deductively?

Two obvious things could happen: the SC could simply be put back to sleep, retaining no memory of anything that just transpired, or all the partial results of the initial association and subsequent polling could somehow be remembered. We will call these two general strategies the pulse model and the pressure model of SC activation. The term "pressure" is intended to suggest a cumulative buildup of evidence in favor of running the SC. The term "pulse" is intended to suggest the transient nature of SC trigger patterns which have no memory of past partial successes; unless all required components of a pulse SC's trigger pattern are found to be true simultaneously, the SC never fires.

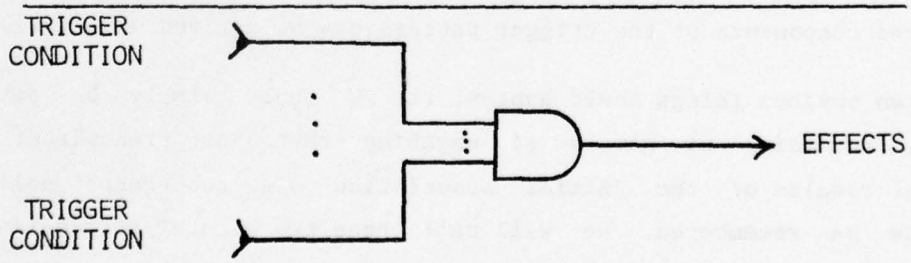
There will be applications where we can get by with a pulse model. Sussman, for instance has what amounts to a pulse model in his electronic

circuit analysis program [SS1]. Since his SCs are monitoring nominally straightforward physical conditions (e.g. the presence or absence of a pulse, the instantaneous current through, or voltage drop across a component, the state of some transistor), the pulse model is apparently adequate in his case.

However, the pressure model is far more interesting theoretically, both because it conserves what it has discovered as partial evidence (via what may have been a very costly deductive process), and because it can be made to give rise to "lingering motivations" within the system, i.e. to focus what happens in the future on the basis of what has happened in the past. We will take a moment to expand on the notions of pulse and pressure by drawing analogies to computer hardware; then we will consider how the pressure model might be implemented and the theoretical implications of doing so.

3.1.1 Pressures, Pulses, AND Gates and Memories

If we were to ignore (without loss of generality) the disjunctive components of an SCs trigger pattern, we would see a pattern of conjunctive conditions. The analogy is thus one of an AND gate, where the gate's inputs are the conjunctive components of the trigger pattern, and the gate's output is the cumulative effect produced on the system by running the SC's body:

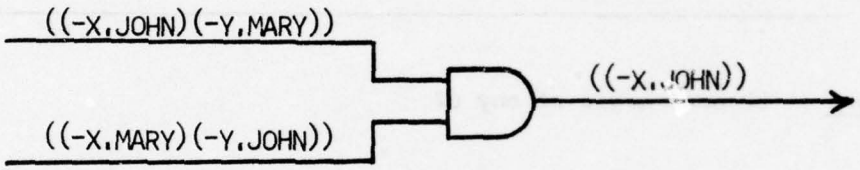


Display 75. An SC as a symbolic AND gate.

But, of course, instead of binary pulses, each line will now carry more complex symbolic information. In our scheme, this symbolic information on any given input line can be distilled down to a binding list, indicating how the variables of the conjunctive element must be bound in order for its input line to be high (active). The sheer existence of the line will imply a specific relationship (e.g. LOVES) among the bindings which flow on the line.

For the analogy's sake, let us assume that each SC body's computation will be such that the body simply asserts a new piece of knowledge whenever its trigger pattern is fulfilled for any particular set of input bindings. (That is, the body produces no strange side-effects.) If we make this assumption, the output of our AND gate can then also be expressed as a binding list. An example is shown below in Display 76.

```
(AND (+ 1 (LOVES -X -Y))        ========>   (UNHAPPY -X)
     (+ 1 (NOT LOVES -Y -X)))
```



Display 76.

Given any population of SCs restricted in this manner, it will be possible to construct a logically equivalent hardware configuration, keeping in mind that lines carry not binary pulses, but binding lists.

Now, "pulse" in this setting will have the obvious interpretation: bindings exist on lines only in short bursts; if not all conjuncts are present simultaneously (and furthermore, have compatible binding values), no output will be produced by the AND gate, and no memory of a near-miss will ever exist.

"Pressure", on the other hand, will amount to levels being maintained for

extended periods of time on the various lines. If this is to occur, we must imagine each line to possess a memory for recording all the various binding lists which have "backed up" on that line, but which have not yet participated in a full triggering of the SC. For example, suppose (for the SC trigger pattern of Display 45), the system acquires:

(LOVES JOHN MARY)
(LOVES PETE RITA)
(LOVES SALLY BILL)

Display 77. Some knowledge that backs up.

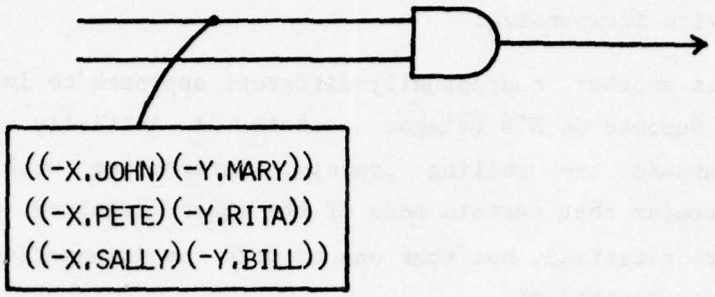
---

all before it becomes aware of any of:

(NOT LOVES MARY JOHN)
(NOT LOVES RITA PETE)
(NOT LOVES BILL SALLY)

Display 78. Information that would unclog the backup.

Then, in the pressure model, the backup on the (LOVES -X -Y) line of the AND gate of Display 76 would look like:



Display 79. Lovers backups.

It is interesting to contemplate what such a logic would look like, how it would behave, and how it would be implemented in hardware gates that could actually manipulate symbolic signals on *all lines instead of binary pulses*. (What would correspond to a counter or a flip-flop?) However, rather than pursue the hardware analogy, we will turn to software alternatives and their implications for carrying out a pressure model.

3.1.2 SC Splitting

One obvious technique for managing pressure would be simply to attach each backed up binding list to its associated conjunctive component in the trigger pattern. This would be directly analogous to signals backing up on an input line of the AND gate. But, whereas we would hope to solve the time problem of recognizing when backed up signals were finally usable by associative hardware in the AND gate analogy, the runtime overhead incurred by this backing up of signals in a sequential simulation makes it an unreasonable idea. In addition, while backed up binding lists certainly do constitute a memory for what has happened in the past, the memory would be buried in an obscure place and form if pressure were implemented by attaching a list of backed up bindings to components of SC trigger patterns at run time.

Because of this, it is hard to see how to put such a scheme to use as a

source of motivation (to fulfill bindings of the remaining AND gate lines) to the system; and since deriving goal direction from partially activated SCs seems to be an important idea, we would reject this approach to pressure even if it were timewise inexpensive.

But there is another, conceptually different approach to implementing the pressure model. Suppose SC X's trigger pattern is initially stimulated by stimulus A; suppose the polling process, interacting with the deductive mechanism, ascertains that certain ones of the other required components of the pattern are satisfied, but that one or more remain unsatisfied (that is, not all of L(A) is satisfied).

If the parts satisfied by the polling process are Xl,...,Xj, and the unsatisfied ones are Yl,...,Yk, we can conserve much of the effort expended to that point by splitting off the as-yet unsatisfied portion of the pattern, Yl,...,Yk, and instantiating as much of it as possible with the various bindings derived from those parts, A,Xl,...,Xj, which have been satisfied.

In general, this may give rise to numerous partially instantiated patterns, corresponding to each possible binding set (way to perform a partial instantiation of Yl,...,Yk) derived from A,Xl,...,Xj. We will call this process SC splitting.

SC splitting will amount to a narrowing of the scope of applicability of the original SC, S, to some subset of the original class of situations to which S is applicable. In this sense, SC splitting provides not only an implicit memory for what has happened (i.e., it creates simpler AND gates whose existence represents some combination of backed up binding lists), but also a theoretically important strategy for narrowing the system's future focus according to what it has become aware of in the past.

As an example, suppose there is an SC whose pattern is:

```
(AND (+ 1 (DISLIKES -X -Y))                ====>  (DENY -X -Y)  **
     (+ 1 (REQUESTS-HELP-FROM -Y -X)))
```

Display 80.

** Again, we are using rather ludicrous predicates in these examples simply for the sake of brevity in the representation!

---

and the stimulus: (DISLIKES JOHN PETE) appears. Suppose furthermore that this SC, aroused by this stimulus, is then unable to determine (REQUESTS-HELP-FROM PETE JOHN) during polling. Rather than allowing the SC simply to doze off again, we will instead split and instantiate it, giving rise to a new SC whose pattern is:

(+ 1 (REQUESTS-HELP-FROM PETE JOHN)) ====> (DENY JOHN PETE)

Display 81. A split and instantiated SC.

---

thereby arming the system with a new piece of knowledge of more limited scope than before and preserving a "memory" of partial activation.

3.1.2.1 SC Splitting, Context and Frames

If we now visualize what begins to happen system-wide in a model that splits SCs, we can see the possibility for many SCs becoming partially aroused and satisfied by some stimulus, say, (DISLIKES JOHN PETE). That sub-population of SCs which nibbles at this stimulus constitutes, in a very real sense, the model's composite understanding of what DISLIKES means; after all, the "meaning" of a (DISLIKES -X -Y) relationship can be no more than the sum of all larger patterns in which it occurs.

The act of splitting and instantiating all SCs that nibble at some stimulus involving DISLIKES therefore amounts to conditioning the system with an implicit DISLIKES relationship between the two people involved. The way the system behaves in the future will then reflect this conditioning through the population of specialist SCs which were spawned by the splitting process.

If we regard the population of SCs which would nibble at some stimulus as the frame (Minsky [M3]) for that stimulus, then the event of splitting and instantiating that population for some particular instance of the stimulus

will correspond to instantiating its frame with specific terminals (to use Minsky's terms). In other words, all instantiated and split SCs which would nibble at (DISLIKES JOHN PETE) will collectively represent a copy of the "DISLIKES" frame, instantiated with respect to JOHN and PETE.

This similarity between partially activated SCs and frames is aesthetically pleasing, since it seems to tie some ideas together. All the ideas have to do with the system being able to "tune" itself automatically to a situation in order to access the most relevant world knowledge in the most efficient manner at the most opportune moment. We have previously been interested in this idea of tuning within a totally different, demand-based computation framework as described in [R2]. In that framework, tuning means planting so-called "bypasses" in the selection networks of the other theoretical half of the CSA system, the more goal-directed components.

Thus, we have a pleasing closure of ideas about context in the demand-based components of the CSA system as well as the spontaneous components, and one which relates both to the general frames point of view.

3.1.2.2 Mechanics of SC Splitting

There are two questions concerning the manner in which splitting is to be controlled, and the manner in which the split results are to be handled. We are in the process of implementing some of the ideas about to be described, but the code included in the appendices does not reflect any of these ideas.

The easier question is: what ought the system to do with SCs which have arisen via the splitting process? There are two obvious alternatives:

(1) plant them in a trigger tree

(2) send their patterns to a "curiosity queue"

Planting split and partially instantiated SCs in a trigger tree amounts to creating a population of specialists who will be geared to more specific events in the arena than the SCs from which they arose.

If split SCs are planted in a separate tree, where ought the tree to be accessed? Because the tree of split SCs represents a more special-case version of certain SCs in the original population, it would seem most reasonable to

attach the new tree to a point on the same channel to which the original tree is attached at a point before the original tree. In this manner the specialist knowledge could have a crack at passing patterns first, blocking them on successful activations so that the generalist knowledge need never see them.

3.1.3 Story Character Followers

In The Magic Grinder story which we have been using to focus our story comprehension research, there is a very natural application of this notion of SC splitting that has to do with modeling the story characters.

As each character is introduced, or as each new characteristic of a character is discovered, we propose to dangle the new features over the population of SC generalists, then catch, split and instantiate all the nibblers to capture or refine the "frame" for the character whose features have been nibbled at. Then we will plant the resulting split SCs in a tree associated with the character. We imagine there to be a tree for each story character, and we call the tree for each character a character follower.

Character follower trees will represent fragments of the generalist knowledge which have been tuned to the idiosyncrasies of each character. Beause of this, they can do a more efficient job in the role of spontaneous inference (including predictions) where the character is concerned; character follower trees hence represent a significant form of search reduction in the processes which will predict what any given character is likely to do, or how he is likely to react to any given situation.

We will have more to say about the use of SCs as a partial basis of inference, but we will also argue that there is a large class of inference not appropriate to this paradigm.

3.1.4 Curiosity Queues

Planting split SCs in new trigger trees is a way of "subconsciously" conditioning the system in a data-driven fashion. We use "subconsciously" here in a figurative sense to suggest that the resulting SCs, while more specific, become simply another component of the background of watchers. In this sense, they contribute no additional motivation to the system.

A curiosity queue, on the other hand, will provide an alternative receptacle for split SCs that can be regarded as a focal point for more "active" motivations of the system, those motivations which are not a priori part of the model, but rather which arise because of curiosity about things only partially revealed, say, in a story.

Suppose, rather than planting a split SC in a trigger tree, we place the unfulfilled parts of its pattern on a queue which will be scanned periodically. Scanning will mean: attempt to answer the lingering questions on the queue, calling upon the general powers of the system's deductive component. In this sense, items on the curiosity queue acquire an elevated status of being actively sought. The queue will define a set of "problems" by which, say, the reader of a story is bothered.

We would argue that there is a very important distinction in theory between this use of split SCs and the replanting technique. Because events in the deductive process, and particularly partial evidence it may uncover, can provide new fuel for still other parts of the SC component, the periodic scan of the curiosity queue will be the origin of an important enriching cycle between the deductive component and the SC component.

When to plant vs. when to send to the curiosity queue seems to be the main question. One obvious idea is to send to the curiosity queue only those split SCs which are almost completely triggered, while merely planting others which are still rather tentative. Another idea would be to <u>preview</u> what the body of the SC would do if run, and if its potential contribution looked especially relevant to the solution of some other problem on the queue, promote the split SC to the curiosity queue rather than interring it in a trigger tree. Previewing would therefore be a technique wherein priority would be given to those SCs with the most apparent promise for contributing to the solution of some other problem. Previewing would require some restrictions on the structure and semantics of SC bodies, or an independent precis (in the CSA declarative representation) of what the SC would do if run.

We are presently contemplating how to implement SC splitting. Apparently, the directives about when and how to split must be contained as features of each SC, or possibly as features of the trigger tree which represents an entire population of SCs. Current thinking is to give each SC a SPLIT-ON

condition which will name the components of the trigger pattern which must be satisfied in order to split. Clearly, there will be cases where any combination of components which pass some criterion should instigate a split, so our syntax will have to be rather flexible. Perhaps it will be necessary to distinguish the conditions for splitting and planting from the conditions for splitting and queuing. In any event, we will hope to discover techniques for factoring the splitting directives out of the individual SCs and into a more central collection of heuristics.

SC splitting and its ramifications are very fertile grounds. But rather than progressing from half-baked ideas to quarter-baked ideas, we want now to turn to some specific theoretical applications of spontaneous computation as a paradigm in cognitive models, particularly those in the domains of language comprehension and problem solving.

3.2 Spontaneous Computation as a Basis of Inference

Inference, as distinguished from deduction, is a process wherein new knowledge, formed out of existing knowledge, arises without solicitation. In deduction on the other hand, new knowledge also arises from existing knowledge, but only after solicitation from some component of the system with a specific need. Therefore, deduction is goal-directed, or "top-down", inference is data-directed, or "bottom-up".

In the CSA model of the more goal-directed processes (described in [R1], [R2] and [R3]), we have been concerned with a special class of inference dealing with actions and knowledge about cause and effect. We call knowledge about cause and effect algorithmic knowledge, and therefore term inferences which arise from this knowledge algorithmic inferences. We will want to make a clear distinction in theory between this class of inference and the class for which, we propose, SCs should serve as a basis.

Algorithmic inferences, by definition, have to do with why actions are performed in a given context and why actors desire certain conditions to be in effect. The point of view is that purposeful actions are done for reasons which can be inferred, given a rich enough knowledge about cause and effect. Similarly, the reasons why a potential actor might desire some condition to be true nearly always relate to the enablement requirements of planned subsequent

actions which will collectively lead to some final goal.

3.2.1 Algorithmic Inference

In [R1], [R2] and [R3], we have argued that this teleological knowledge is highly structured, and accessible in very orderly, "refereed" manners. In brief, the process of algorithmic inference goes as follows: state S or action A is perceived to be (respectively) desired or performed by some actor; the system's goal is to produce a context-sensitive explanation of this desire or action.

In the CSA setting, there is a large number of relatively small cause-effect schema, called abstract algorithms which describe specific ways for causing states and statechanges to occur. The abstract algorithms are organized into so-called causal selection networks, one network for each state and statechange concept known to the system. For example, there is a statechange LOCATION network which serves as the organizing structure for thousands of (in principle, not in the running model!) strategies for changing the location of various types of objects from a starting point to a terminal point. Clearly, the strategy will be dependent on both the objects and places involved in the statechange of location, who it is that will be effecting the strategy, and a general awareness of the context in which the strategy will be effected. It is the purpose of a causal selection network to ask an orderly progression of questions to illuminate as much of the relevant information as required about the situation so that an intelligent selection of one strategy from among the large number of contenders which may exist at the terminals of the selection network can be made.

The causal selection networks therefore carry out the theoretical point of view that intelligent selection from among a collection of known alternatives is a primary aspect of human intellectual abilities.

But if the system has an algorithmic base of thousands of cause-effect schema, each tuned to a small part of the world, we can also confer upon it the reverse ability to determine of any given state or action, X, where X could conceivably participate in cause-effect strategies. Knowing where X could participate, it is then possible to trace upwards (backwards) through layers of CSA patterns and causal selection networks from the set of starting

points (strategies) in which X might be participating. By applying the questions in the network as this upward climbing occurs, it seems to be possible to rule out most possible "interpretation paths" quickly, because of failures of the situation in which X participates to agree with the tests in the network. (This is described in more detail in [R2].)

The final interpretation, i.e., the algorithmic inference, from X is that path (or collection of paths) which survive long enough to connect up with a prediction ** which has been made concerning the actor associated with desire or action X.

** The CSA system's predictions at any given moment have been derived from other CSA patterns, primarily ones involving the so-called inducement and motivation links. These are described in [R3].

We propose that this type of inference, inasmuch as it interacts with a highly structured knowledge about cause and effect, ought not to be modeled by spontaneous computation. Without the tremendous search- restricting and mediating influence of the causal selection networks, the system would explode combinatorially with possible interpretations of actions and desires in context. Fundamentally, SCs are local entities which are independent of one-another. They are inherently resistant to organization into the kinds of larger structures which seem to be appropriate for cause and effect knowledge, and we believe that it would be incorrect to attempt to cast them in this mold by building "spontaneous computation selection networks". SCs are simply not intended to be selected among, and are hence intrinsically ill-suited for algorithmic inference.

3.2.2 Non-Algorithmic Inference

What other types of inference are there after we eliminate cause-effect based inference? Clearly, a lot! But it is very difficult to characterize under any one banner what is left over. Common to all remaining types, however, will be a characteristic absence of intentionality. We feel that it is to this residue of non-volitional inference types that SC-based inference ought to be limited... to actionless situations, "settings", which convey

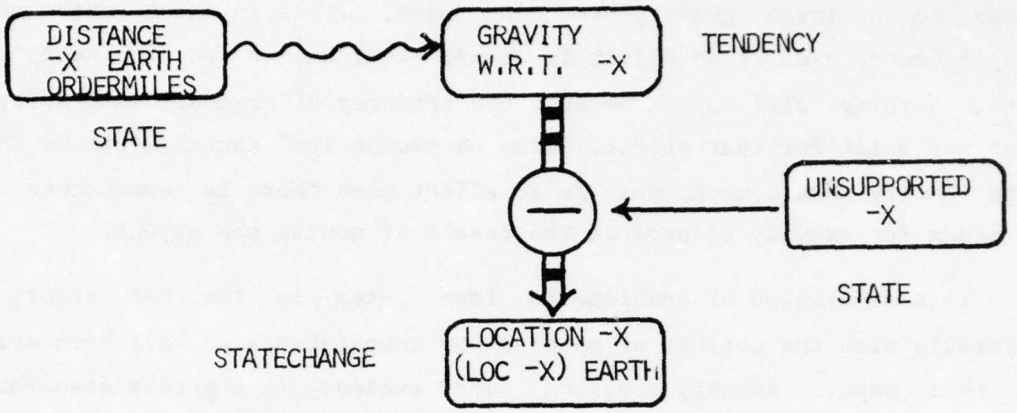
information via state descriptions which have not been purposefully caused by actors. This will embrace things like descriptions of characters in stories, scenes laying out spatial or emotional relationships, patterns delimiting how a story character might be expected to behave in general situations, and so forth. In other words, SC-based inference ought only to deal with non-purposeful aggregates of states which are the way they are for no particular reason, but which nevertheless will represent an often rich basis for inference.

3.3 Spontaneous Computation in a Plan Synthesizer

3.3.1 SCs as Models of CSA Tendencies

In the CSA model, there are five theoretical types of events: actions, states, statechanges, wants, and tendencies [R1]. A tendency is defined to be an action-like event, in the sense that it causes new states and statechanges, but an event in which there is no animate actor. A tendency is therefore non-intentional force which must occur whenever its set of enabling conditions occurs.

For example, our commonsense notion of gravity tells us that whenever an object X is in an unsupported state and close to a very large mass, it will begin changing its location from where it is toward the large mass. In CSA syntax, we write this as follows:



Display 82. Commonsense Gravity

CSA distinguishes the notions of enablement and gating. An enablement is a condition (state) which must be in effect in order for an action to proceed, regardless of what the action is intended to accomplish. For example, if one wishes to pick up an object, he might grasp after having ensured that his hand is hovering around the object he intends to pick up. In this situation, regardless of whether or not the gating state (AROUND (HAND P) OBJECT) has been satisfied, in order for the action GRASP even to begin, the enabling condition

(MOBILE (FINGERS (HAND P)))

Display 83.

must be true, and must remain true for the duration of the GRASP movement. Enablements therefore are associated with the context-free requirements of actions in isolation, while gates describe the context in which the action must be performed in order to achieve some desired result.

There is an equally natural division of enablements and gates in CSA tendencies. The enabling conditions for a tendency, T, are those states which must be true in order for T to exist. For gravity, we adopt the view that this

force simply does not exist with respect to some object unless that object is close to a large gravity-producing mass. (This is the commonsense notion only!) Hence, even if an object is unsupported, unless it is near a large mass, nothing will happen because the tendency of commonsense gravity simply does not exist for that object. Gates on tendencies' causal links on the other hand specify what context must be in effect when there <u>is</u> commonsense gravity in order for gravity to produce the result of moving the object.

This separation of enablements from gates in the CSA theory meshes naturally with the notions of spontaneous computations we have been developing in this paper. Roughly speaking, every tendency is a purely associative, or state-based inference which we are obligated to make whenever we can. Thus for example, we may model the tendency COMMONSENSE-GRAVITY by an SC whose trigger pattern might appear as:

(AND (+ 1 (DISTANCE -X EARTH ORDERMILES))
(+ 1 (UNSUPPORTED -X)))

<u>Display 84</u>.

But somehow, the two conditions of Display 84 do not have equal status; the support status of an object is, conceptually at least, far more apt to vary than the object's distance from EARTH, and besides, the distance from earth governs gravity's <u>existence</u> (with respect to that object), whereas support relations govern its <u>effects</u>. Since the CSA theory categorizes these two conditions differently, why not reflect this in the SC implementation of gravity?

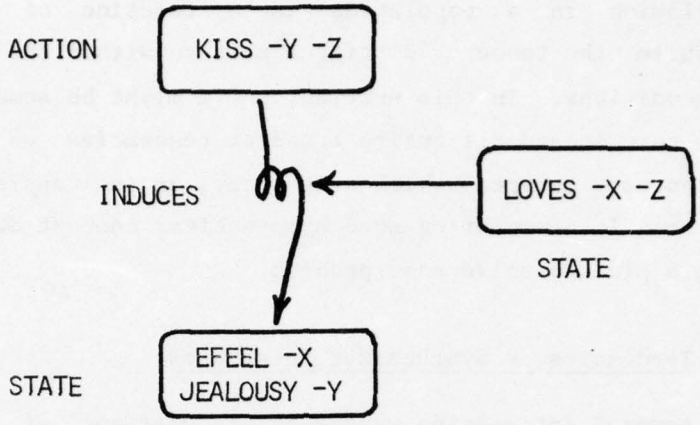
We can do so by retaining only the second condition about unsupportedness as the SC's trigger pattern, and placing the SC itself in a tree which models an entire population of SCs which share the DISTANCE condition as an existence enablement. Then, by turning the entire tree on and off on the basis of knowledge about our distance to the earth, we will be able to model large-scale shifts in context (e.g., leaving the earth's influence) quite naturally.

We therefore imagine populations of tendencies modeled by trees of SCs. A tendency's inclusion in a population is a function of the tendency's enablements, while the tendency's trigger pattern within the tree is derived from its gate conditions. In this setting, there might be some SCs whose sole job would be to turn on and off entire trees of tendencies on the basis of large scale context changes which arise, say, as the result of executing a plan, or as the result of entering some hypothetical context during the course of synthesizing a plan to solve some problem.

3.3.2 SC-based Tendencies as Synthesizer Interrupts

There are several interesting ways such populations of tendencies can interact with a problem solver. One obvious way is as a kind of bookkeeper; when the plan synthesizer ungrasps an object in mid-air, some tendency should arise and inform the synthesizer that the object is now in a state of motion toward the ground. Another computation might then be awakened to compute its expected trajectory and landing point. The synthesizer, meanwhile, may either have decided to alter its plan in a way which would disable the (undesired) tendency by ungrasping the object only after ensuring the object was first supported. Or the synthesizer may decide to ignore the falling condition. In that case, should the object be required later, the database will at least contain a prediction about where the object might have gone (i.e. landed).

Clearly, there are tendencies which are not exclusively physical in nature; there are emotional and social tendencies as well. We are currently pondering the relationship between CSA patterns which involve the so-called inducement link and non-algorithmic inference. For example, suppose we wish to express the emotional principle: if X sees Y kiss Z, someone X loves, X is liable to feel jealousy toward Y. CSA provides the expressive declarative power we need:



Display 85. How to induce jealousy.

In an operational sense, this piece of knowledge is highly algorithmic, even though it deals with (unexplainable?) human emotions: if John wants to make Bill jealous, one strategy might be to find someone Bill loves, then kiss her in front of John. On the other hand, there is a good case for regarding this pattern as a tendency, since, on another occasion it might not be intentional at all, but something that happens by chance.

Actually all tendencies are like this inducement pattern, in that they can play either a goal-directed role or a spontaneous role. For example, there are times when we may wish to regard gravity algorithmically by consciously employing it as the top-level strategy for moving an object, and there are times when we would want it to arise associatively. Because of this, our inclination now is to regard patterns like this, and indeed, all tendencies as both algorithmic and spontaneous. In the current plan, tendencies and inducement patterns will be integrated into both components of the system: they will be stored as terminals of causal selection networks, and they will be planted as SCs in trigger trees. Although there is still some ambient confusion here, we feel that the dual role approach is inescapable.

3.3.3 Subgoal Protection

When confronted with a goal, the CSA plan synthesizer (1) accesses the causal selection network associated with the predicate used to express the

goal, (2) descends downward through the net, asking questions about the nature of the goal and about the environment, following the path prescribed by the answers to these questions, then (3) adopts the strategy specified by the CSA pattern it has located at the net's terminal node. Adopting a strategy means (a) instantiating the pattern schema with the particulars of the goal it is to solve, (b) committing to particular things (these are the hammers, cups, verbal phrases, etc.) that will be required in order to carry out the strategy... the components of the strategy introduced by adopting the strategy. Finally, the synthesizer (4) solves any unmet subgoals in the strategy via recursive calls to itself.

Often, it will happen that subgoals are not compatible, or at least incompatible with respect to some particular ways of solving them. Thus, as it has been recognized for some time now, there is the danger of a purely recursive plan synthesizer first solving a subgoal, but in the synthesis of the next subgoal, doing something that would destroy the effects presumed to have been achieved by the first subgoal's solution. Since subgoals are states of the world which usually must be in effect simultaneously at plan execution time, if solving one destroys another, the plan simply won't work. Sussman has called this problem "prerequisite clobbers brother goal", and it was precisely this problem which motivated much of his dissertation research [S1]. We will refer to this problem as "subgoal annihilation" [L1].

Subgoal annihilation may happen for one of two reasons: either the CSA pattern the synthesizer has adopted as a strategy is conceptually faulty (e.g., it may have been incorrectly learned, or may have been learned and works for one case, but not for another), or, the pattern is not conceptually faulty, but the ways the synthesizer has gone about solving the subgoals are antagonistic to each other.

In the former case, restructuring of the strategy pattern itself is called for (London [L1] is working on this problem in the context of the CSA system); in the latter case, retrying the synthesis of the subgoal, this time avoiding problem spots (and noting them for the future), is called for.

The CSA group is just beginning to address these problems beyond what Sussman did, and we have no firm ideas yet. Nevertheless, the SC component of the CSA system is presently capable at least of detecting subgoal annihilation

in the following manner. As each subgoal is addressed and solved, the CSA synthesizer "protects" it by creating an SC which will scream and jump up and down in reaction to other patterns generated by the synthesizer which are inimical to the protected subgoal. Currently, this amounts simply to watching for the erasure of the subgoal which has been imagined to have been solved (and hence stored as a fact in the hypothetical context in which the corresponding level of synthesis occurred). However, the protect pattern could (and should) be made sensitive to a broader spectrum of inimical patterns within the existing SC framework (e.g. via OR conditions in the protecting SC's trigger pattern).

When all subgoals have been successfully solved, their protecting SCs are destroyed, any actions which the subgoals are intended to enable or gate are added to the execution run stream being generated, then the current level is assumed to have been solved. At a higher level, an SC will be created to protect this goal for as long as necessary, and so on, until the original problem has been solved.

3.3.4 SCs as Constraint Violation Interrupts

As the plan synthesizer gets deeper and deeper into some plan, it will create more and more SCs to protect subgoals. This amounts to a growing minefield of constraints. Although such constraints arise for the purpose of protecting subgoals, it is possible to interpret them in another fruitful way.

For example, suppose the synthesizer has generated a plan wherein AGENT grasps an immovable object, effectively tethering himself to within a small neighborhood of the immovable object. Suppose then that the next subgoal involves moving across the room, out of the tether range.

At that point, one of two things could happen: either the synthesizer could be aware that AGENT was grasping the immovable object, and attempt to have AGENT ungrasp it (probably leading to a subgoal annihilation complaint from the protecting SC), or it could have AGENT simply proceed with the WALK plan to move across the room.

Now, if at the time the GRASP had been achieved and protected, another SC had also been planted with a pattern which would react whenever AGENT moved

out of the tethered range, as the WALK plan was generated, this SC could inform the synthesizer that it had just violated a constraint.

Detecting such a constraint violation is important for two reasons. First, it will alert the synthesizer that it is generating a faulty plan. But second, and perhaps more important, it can provide the synthesizer with a situation in which it may be able to learn something. In this example, what might be learned is that whenever a statechange in location of self is being planned the synthesizer should first ask whether or not AGENT is tethered to an immovable object. If the answer is "yes", it should employ a strategy that includes detaching AGENT from the object before solving the statechange in location.

In the CSA system such an act of learning would amount to implanting a new test in the causal selection network for causing statechanges in location (described in [R2]). By planting a test about "tetheredness", the system, in a very important sense, extends its conception of what is relevant to the successful solution of statechanges in LOCATION.**

** It might seem alarming to force the system into having to ask "is AGENT tethered to an immovable object" every time it wishes to synthesize a plan for moving. However, because the CSA causal selection networks have an ability to bypass portions of themselves (described in [R2]), such a question need be asked only once for any given agent. Because of the bypass which becomes implanted, the question will not be seen as part of the network from that point until some grasp of an immovable object actually occurs. At that time, the bypass would be destroyed, making the statechange LOCATION causal selection network once again sensitive to "tetheredness". Thus, although the system has learned that tetheredness is a factor in location changes, it will not have "consciously" to think "is AGENT tethered to an immovable object" every time it generates a plan for AGENT!.

Another related class of constraint violation is typified by "the hand in the gumball machine" dilemma: a kid wishes to remove a prized object from a container with a small opening. The opening is just large enough to admit a limp hand, but is too small for a fist, or a "fat" hand to pass through. The kid, employing the standard grasp strategy, will experience a constraint violation (in this case, at run time) when he attempts to remove his grasping fist from the container; namely, he will realize that his fist is stuck! From this violation, he might be able to learn two interesting principles: (a) that a side-effect (byproduct in CSA terms) of a GRASP action is that the hand

becomes "fat", something that may never have occurred to the kid before, and (b) that whenever he is about to synthesize a plan to change the location of his hand (on a path through an opening), he should first ask whether or not he will have satisfied the important enabling condition for the move: HAND will fit through the opening! (a) will amount to learning a new piece of causal information, in the sense that it augments an existing CSA pattern about grasping. (b) will amount to learning that a precondition for moving a hand is that the enablement "hand fits through constrained places on the trajectory".

3.3.5 SCs as Plan Optimizers

If we are concerned only with synthesizing plans which are execution-time consistent, but which are not necessarily conservative of materials, time or energy, there need be little cross-communication among the synthesizer calls for the various subgoals of a main goal. In this setting, it may just happen that the leftover products of one subgoal's solution will aid in the solution of the next subgoal. But in general such interactions will be coincidental. It would clearly be more desirable to build some notion of optimization into the control of the synthesizer.

SCs can be put to use as optimizers by a technique we call *previewing*. If, as it commits itself to a top-level strategy via traversing a causal selection network, the synthesizer "peeks ahead" a level or two to preview what subgoals will eventually require solutions, it can create SCs to watch for patterns which would in some way be of use to the solution of each previewed subgoal.

Typically, useful patterns will be those describing states which could serve as enabling or gating conditions for an action whose execution was anticipated. For example, suppose the synthesizer previews and determines that it will eventually have to synthesize a plan wherein the light switch on the wall is turned off. By, say, a one-level preview, it might determine that among others, one enabling condition for such an action is that AGENT be near the switch. If before it begins solving any other subgoals, the synthesizer creates an SC to monitor for the condition "AGENT is near the switch", then any subgoal whose solution incidentally brings AGENT near the switch will trigger the optimizing SC.

The optimizing SC could then interrupt the synthesizer, informing it that, since it was in the neighborhood, it might consider solving the rest of the switch-off problem. The synthesizer could then either reject the suggestion, or do it, running the obvious risk of annihilating some other condition prematurely (e.g., the subgoal whose solution was interrupted requires light in the room in order to proceed). But presumably such annihilations are being monitored by subgoal protectors who would complain at still another interrupt level, and so forth.**

** There is a question about whether this type of optimization belongs in the conceptual planning stage of plan synthesis, or in the execution phase. In the execution phase, the optimization would occur over an action sequence rather than by the previewing mechanism called for in the synthesis phase. The problem is analogous to the problem of whether an optimizing compiler should do its optimizing at the code generation level (or at a still higher algorithmic level), or simply shuffle the code around after it has been generated, without regard for the semantics of the situation. Although we have no final arguments in its favor, we are biased toward optimization at the conceptual synthesis level.

3.4 SCs as Hierarchical Situation Characterizers

It is natural to think of SCs in terms of numerous populations, each population tuned either to specific facets of the environment, to specific contexts, or to specific levels of resolution. In other words, SCs can be structured and put to use hierarchically. They can be regarded either as data-driven or as goal-driven, because of the way they interact with the deductive components of the system during the polling process.

Hierarchically structured populations of SCs can be put to use in interesting ways to convert context-free information at the data level into progressively higher, more semantic and context-dependent assessments of a situation. Consider a chess game. Context-free information in a chess game presumably has forms such as: "PAWN1 is attacking KNIGHT2", "ROOK2 is not in immediate danger", "QUEEN has the following three moves", and so forth. The context free information is that which can be gathered on a very local and mechanical, piece by piece basis, with no regard for its contextual implications.

We now imagine a population of SCs whose job it is simply to watch this

level of characterization. The trigger pattern of a typical SC in this population will be a mixture of context-free parts and more semantic and contextual parts, perhaps things like "does the bishop play a role in constraining the opponent's rook?" Now, when the SC triggers on the basis of very syntactic information, it will (via the polling process) pose the as yet unanswered semantic and contextual questions in its trigger pattern to the deductive component of the system.

The deductive component, being fishbowled by yet other SCs, will pose a new generation of questions designed to answer this question, possibly giving rise to new SC invocations. Thus, regardless of the answer's outcome, the very fact that the original SC posed the question can give rise to an upward spiraling of more semantic awarenesses about what is happening on the board. At another higher level, we would imagine there to be more abstract SCs designed to react to lines of constraint, lines of force, mounds of power, or whatever. Presumably, these top level characterizers would correspond closely with the concepts a chess expert employs.

This deceptively simple notion - mixing the syntactic with the semantic and contextual in SC triggr patterns - seems to be the key to many problems of context. It provides the system with a starting point at which to begin making semantic conjectures; these conjectures, whether or not they prove to be true, can be important catalysts for higher level SCs via questions sought during the deductive processes. In this sense, SCs provide a significant source of "upward awarenesses" which at some point hopefully make contact with the strategy, algorithmic, or goal-directed levels of the system.

3.5 Other possible arenas

In this paper, we have considered only one of many possible arenas for spontaneous computation: LISP S-expressions moving from logical point to logical point in a model. Clearly there are many other arenas. Two notable ones at which we will take a brief look have to do more with control than with data. The first idea belongs to the LISP machine group [G1] and the KRL group [BW1], and the KRL group calls it procedural attachment. The second idea has to do with spontaneous computation which is triggered on the state of control rather than on the state of data.

3.5.1 Procedural Attachment

Procedural attachment (PA) factors the problem of SC quite a bit differently from the way we have been viewing it. In PA, code is triggered by the act of referencing an entity in the model. Most generally, "reference" could encompass any act of attaching, detaching, or inspecting information associated with an entity in the system; it could mean "the address of the entity appears in an active register (accumulator) of the computer". In a LISP machine (hardware) style of PA, reference means, e.g., taking the CAR or CDR of a cell, or changing the CAR or CDR of a cell, or requesting the functional definition of an atom. In the KRL use of the concept, PA is a way of factoring intelligence out of code which manipulates objects and into the objects the code manipulates. For example, if there are numerous logical types of entities in the system, each requiring a different style of formatting when printed out via a LISP PRINT, rather than coding into PRINT a knowledge of all the various formats, each object type bears knowledge about how to behave when it becomes involved in a PRINT operation. In other words, printing an entity becomes as simple as pointing at the object to be printed and saying PRINT YOURSELF!

There are some interesting possibilities for PA. It represents a sort of distributed SC. It is a style of SC in which each object in the model can be given a priori expectations about the larger events in the system in which it might participate. Because PA distributes intelligence, it is perhaps modular in the way we imagine the human brain to be.

One possibility would be to move from the simple PA concept of reference to a more sophisticated concept of "occurrence in a pattern". In this type of PA triggering, a referenced entity's attached code would run only when the entity occurred in some special role in some larger pattern. But SURPRISE! This brings us back to the style of SC we have already considered. In fact, if we have PRINT "point at" an object by placing a request (PRINT <obj>) on some channel, we can couch the whole notion of PA in our style of SC.

3.5.2 State of Control Triggered SC

In state of control (SOC) triggering, we specify calling sequences (i.e. configurations of LISP's control and value stacks) which are of interest;

whenever such a calling sequence occurs, the system will be interrupted and the code associated with the SOC SC run. An example would be: interrupt the system whenever function GLOP calls function FOO with a second argument that meets criterion X, and FOO then calls BAZ.

London has implemented such a scheme that permits one to specify an arbitrary number of such SOC patterns. London employs the notion of a "call tree" to keep track of partial SOC activations. (The approach is a generalization of the MACLISP "wherein" debugging feature.) Current plans call for using this SOC component in his research in learning; the LISP control which occurs as the base CSA model operates will be monitored by SOC SC's who will notice when interesting or unexpected calling sequences occur, interrupt, rummage through the control and value stacks to try to determine what was happening, and then... The suspense is exciting.

4. CONCLUSION

There are no conclusions yet. We are beginning to explore ways to use SC in story comprehension and problem solving. All we know now is that SC underlies quite a few interesting processes of intelligence. How, where, and when it interacts with goal-directed computation are still mysterious, but hopefully not as mysterious as before.

REFERENCES

[BW1] Bobrow, D., and Winograd, T., An Overview of KRL, a Knowledge Representation Language, Xerox Palo Alto Research Center, 1976

[C1] Charniak, E., Toward a Model of Children's Story Comprehension, doctoral dissertation, M.I.T. AI Memo 266, 1972

[D1] Davies, D. J. M., Poplar 1.5 Reference Manual, University of Edinburgh, TPU Report No. 1, 1973

[DK1] Davis, R., and King, J., An Overview of Productions Systems, Stanford AIM 271, 1975

[G1] Greenblatt, R., et. al., The LISP Machine, M.I.T. Working Paper, 1975

[H1] Hewitt, C., Procedural Embedding of Knowledge in PLANNER, Proc. 2IJCAI, London, Sept. 1971

[H2] Hewitt, C., Viewing Control Structures as Patterns of Passing Messages, M.I.T. AI Working Paper 92, 1976

[K1] Knuth, D., The Art of Computer Programming, Fundamental Algorithms (Vol. 1), Addison Wesley, 1973

[L1] London, P., Abstraction Mapping and Learning in a Problem Solving Environment, doctoral dissertation proposal, University of Maryland, 1976

[M1] Marcus, M., Wait-and-See Strategies for Parsing Natural Language, M.I.T. AI Working Paper 75, 1974

[M2] McDermott, D., Very Large PLANNER-like Data Bases, M.I.T. AI Memo 339, 1975

[M3] Minsky, M., A Framework for Representing Knowledge, in The Psychology of Computer Vision, P. Winston (ed.), McGraw Hill, 1975

[MS1] McDermott, D., and Sussman, G., The CONNIVER Reference Manual, M.I.T. AI Memo 259a, 1974

[NS1] Newell, A., and Simon, H., Human Problem Solving, Prentice-Hall, 1972

[R1] Rieger, C., The Commonsense Algorithm as a Basis for Computer Models of Human Memory, Inference, Belief and Contextual Language Comprehension, Proc. Theoretical Issues in Natural Language Processing Workshop, M.I.T., 1975

[R2] Rieger, C., An Organization of Knowledge for Problem Solving and Language Comprehension, Artificial Intelligence, vol. 7, no. 2, 1976

[R3] Rieger, C., The Representation and Selection of Commonsense Knowledge for Natural Language Comprehension, Proc. Georgetown University Linguistics Roundtable, 1976

[R4] Rulifson, J., et. al., QA4, A Language for Writing Problem-solving Programs, Proc. IFIP Congress, 1968

[RG1] Rieger, C., and Grinberg, M., The CSA Mechanisms Simulation System, University of Maryland TR (forthcoming), 1976

[RS1] Reboh, R., and Sacerdoti, E., A Preliminary QLISP Manual, SRI AI Center Tech. Note 81, 1973

[S1] Sussman, G., A Computational Model of Skill Acquisition, American Elsevier, 1975

[SS1] Sussman, G., and Stallman, R., Heuristic Techniques in Computer Aided

Circuit Analysis, M.I.T. AI Memo 328, 1975

[SWC1] Sussman, G., Winograd, T., and Charniak, E., MICRO-PLANNER Reference Manual, M.I.T. AI Memo 203a, 1971

[T1] Teitleman, W., Interlisp Reference Manual, Xerox Palo Alto Research Center, 1974

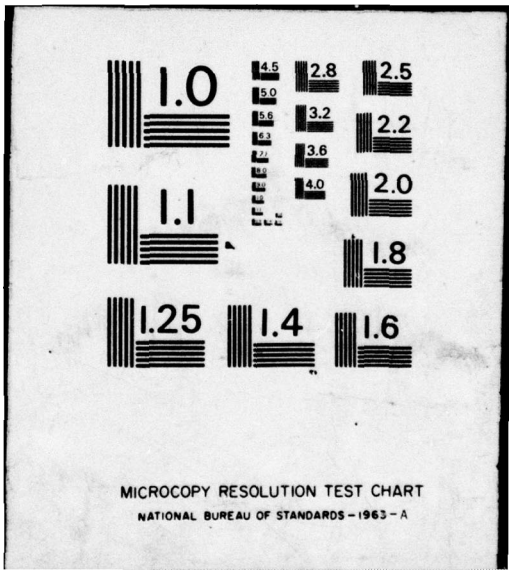
[T2] Tesler, L., et. al., The Lisp70 Pattern Matching System, Proc. 3IJCAI, Stanford, Ca., 1973

AD-A032 812 MARYLAND UNIV COLLEGE PARK DEPT OF COMPUTER SCIENCE F/G 12/1
SPONTANEOUS COMPUTATION IN COGNITIVE MODELS.(U)
JUL 76 C RIEGER N00014-76-C-0477
UNCLASSIFIED TR-459 NL

2 OF 2
AD
A032812

END

DATE
FILMED
1-77


1.0
4.5
5.0
5.6
6.3
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.8
1.1
1.25
1.4
1.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS - 1963 - A

Appendix A - SC Code

```
(DEFUN $MAKE-SC-VAR (PATH)
   ($MAKE-CSA-VAR (CONS 'X (REVERSE PATH))))

(DEFUN $PLANT1 (P TREE PATH VARS)
 (CONS
  (COND ((ATOM P)
         (COND (($IS-CSA-VAR P)
                ($MAKE-VAR (COND ((CDRASSOC P VARS))
                                 (T (PUSH (CONS P ($MAKE-SC-VAR PATH))
                                          VARS)
                                    ::SC-FREE-VAR))
                           TREE PATH))
               (T ($MAKE-CONST P TREE))))
        (T (SETQ TREE ($MAKE-COMPLEX (LENGTH P) TREE))
           (MAPC-N (LAMBDA (PART POS)
                     (SETQ TREE ($PLANT1 PART TREE (CONS POS PATH) VARS))
                     (SETQ VARS (CDR TREE))
                     (SETQ TREE (CAR TREE)))
                   P '(!1 !2 !3 !4 !5 !6 !7 !8 !9))
           TREE))
        VARS))

(DEFUN $LOCATE (PATTERN TREE)
  (AND (CSETQ ::T1 ($LOCATE1 PATTERN TREE NIL NIL)) (CAR ::T1)))

(DEFUN $LOCATE1 (P TREE PATH VARS)
 (COND ((CSETQ ::T1
  (COND ((ATOM P)
         (COND (($IS-CSA-VAR P)
                (CDRASSOC (COND ((CDRASSOC P VARS))
                                (T (PUSH (CONS P ($MAKE-SC-VAR PATH))
                                         VARS)
                                   ::SC-FREE-VAR))
                          ($VARS-OF TREE)))
               (T (CDRASSOC P ($CONST-OF TREE)))))
        ((SETQ TREE (CDRASSOC (LENGTH P) ($COMPLEX-OF TREE)))
         (PROG ((POS '(!1 !2 !3 !4 !5 !6 !7 !8 !9)))
           LOOP (COND ((NULL (SETQ TREE ($LOCATE1 (CAR P) TREE
                                                  (CONS (CAR POS) PATH) VARS)))
                      (RETURN NIL)))
                (SETQ VARS (CDR TREE)) (SETQ TREE (CAR TREE))
                (COND ((SETQ P (CDR P)) (SETQ POS (CDR POS))
                                        (GO LOOP))
                      (T (RETURN TREE)))))
        (T NIL)))
       (CONS ::T1 VARS)) (T NIL)))

(DEFUN $MAKE-COMPLEX (LEN TREE)
   (COND ((CDRASSOC LEN ($COMPLEX-OF TREE)))
         (T (RPLACA (CDDR TREE) (CONS (CONS LEN
                                            (CSETQ ::T1 ($SPROUT TREE)))
                                      (CADDR TREE)))
            ::T1)))

(CSETQ ::SC-FREE-VAR 'FREE)

(DEFUN $CONST-OF MACRO (X) (LIST 'CAR X))
(DEFUN $VARS-OF MACRO (X) (LIST 'AND (LIST 'CADR X)
                                     (LIST 'CDADR X)))
(DEFUN $COMPLEX-OF MACRO (X) (LIST 'CADDR X))
(DEFUN $POS-OF MACRO (X) (LIST 'CAADR X))
```

```
(DEFUN $PARENT-OF MACRO (X) (LIST 'CDDDR X))
(DEFUN $SCS-OF MACRO (X) (LIST 'CDDR X))
(DEFUN $N-TUPLE-OF MACRO (QUERY) (LIST 'CADDR QUERY))

(DEFUN $MAKE-CONST (CONST TREE)
  (COND ((CDRASSOC CONST ($CONST-OF TREE)))
        (T (RPLACA TREE (CONS (CONS CONST (CSETQ ::T1 ($SPROUT TREE)))
                              (CAR TREE)))
           ::T1)))

(DEFUN $SPROUT (TREE) (CONS NIL (CONS NIL (CONS NIL TREE))))

(DEFUN $MAKE-VAR (VAR TREE PATH)
  (COND ((AND (NULL (CADR TREE))
              (RPLACA (CDR TREE) (LIST ($MAKE-SC-VAR PATH)))
              NIL))
        ((CDRASSOC VAR ($VARS-OF TREE)))
        (T (RPLACD (CADR TREE) (CONS (CONS VAR
                                           (CSETQ ::T1 ($SPROUT TREE)))
                                     (CDADR TREE)))
           ::T1)))

(DEFUN $NIBBLERS (P TREE BINDS)
 (APPEND
  (COLLECT ($VARS-OF TREE)
    (LAMBDA (V) (COND ((EQ (CAR V) ::SC-FREE-VAR)
                       (CONS (CDR V)
                             (CONS (CONS ($POS-OF TREE) P) BINDS)))
                      ((EQUAL (CDRASSOC (CAR V) BINDS) P)
                       (CONS (CDR V) BINDS))
                      (T NIL))))
  (COND ((ATOM P)
         (COND ((CSETQ ::T1 (ASSOC P ($CONST-OF TREE)))
                (LIST (CONS (CDR ::T1) BINDS)))
               (T NIL)))
        ((CSETQ ::T1 (ASSOC (LENGTH P) ($COMPLEX-OF TREE)))
         (PROG (CANDS) (SETQ CANDS (LIST (CONS (CDR ::T1) BINDS)))
           LOOP (SETQ CANDS
                      (INDEX (INTO CANDS (LAMBDA (C)
                                           ($NIBBLERS (CAR P) (CAR C) (CDR C))))
                             NIL APPEND))
                (COND ((NULL CANDS) (RETURN NIL))
                      ((SETQ P (CDR P)) (GO LOOP))
                      (T (RETURN CANDS)))))
        (T NIL))))

(DEFUN $SHOW FEXPR (TREE) ($SHOW1 ($GET TREE 'TREE) 0 NIL))
(DEFUN $SHOW1 (TREE IN POS)
  (COND (($IS-SC-LIST TREE)
         (MAPC ($SCS-OF TREE)
               (LAMBDA (D) (INDENT IN) (PRINT D))))
        (T (MAPC-N (LAMBDA (PART HANDLER)
                     (MAPC PART (LAMBDA (PAIR)
                                  (INDENT IN)
                                  (PRINT (EVAL HANDLER))
                                  ($SHOW1 (CDR PAIR)
                                          (PLUS IN 3) NIL))))
           (LIST (CAR TREE) (COND ((CADR TREE) (SETQ POS (CAADR TREE))
                                               (CDADR TREE))
                                  (T NIL))
                 (CADDR TREE))
           '((CAR PAIR) (LIST POS '= (CAR PAIR))
             (LIST (CAR PAIR))))
           NIL)))

(DEFUN $IS-SC-LIST MACRO (X) (LIST 'EQ (LIST 'CADR X)
```

```
                                                    'CQUOTE $C)))

(DEFUN $WEED (NAME LEAF)
   (PROG (PARENT)
     (COND ((NULL LEAF) (RETURN NIL)))
     (RPLACD (CDR LEAF) (ASSOC-DELQ NAME ($SCS-OF LEAF)))
     (COND (($SCS-OF LEAF) (RETURN ($SCS-OF LEAF))))
     (SETQ PARENT (CAR LEAF))
     LOOP (OR (AND (RPLACA PARENT (RASSOC-DELQ LEAF ($CONST-OF PARENT)))
                   ::DONE)
              (AND (CADR PARENT)
                   (RPLACD (CADR PARENT)
                           (RASSOC-DELQ LEAF ($VARS-OF PARENT)))
                   ::DONE)
              (RPLACA (CDDR PARENT)
                      (RASSOC-DELQ LEAF ($COMPLEX-OF PARENT))))
          (COND ((AND ($NULL-LEAF PARENT)
                      (SETQ PARENT ($PARENT-OF (SETQ LEAF PARENT))))
                 (GO LOOP))
                (T (RETURN LEAF)))))

(DEFUN $NULL-LEAF (LEAF)
   (NOT (OR ($CONST-OF LEAF) ($VARS-OF LEAF) ($COMPLEX-OF LEAF))))

(DEFUN $FRAGMENT (P OTHERS)
   (COND (($IS-TRIGGER-PART P) (LIST (CONS P OTHERS)))
         ((NOT (MEMQ (CAR P) '(AND OR ANY))) NIL)
         (T (INDEX (INTO (CDR P)
                         (COND ((EQ (CAR P) 'AND)
                                (LAMBDA (C) ($FRAGMENT C (APPEND
                                                           (ALLBUT C (CDR P))
                                                           OTHERS))))
                               ((EQ (CAR P) 'ANY)
                                (LAMBDA (A) ($FRAGMENT A
                                                (CONS (CONS 'ANY (ALLBUT A (CDR P)))
                                                      OTHERS))))
                               (T (LAMBDA (D) ($FRAGMENT D OTHERS)))))
             NIL APPEND))))

(DEFUN $IS-TRIGGER-PART (X) (AND (NOT (ATOM X)) (EQ (CAR X) '+)))

(DEFUN $PLANT (PATTERN BODY TREE . OTHER)
   (PROG (LEAF VARS (NAME (CDRASSOC 'N OTHER))
          (PRIORITY (OR (CDRASSOC 'P OTHER) 'FRONT))
          (RUNCOND (OR (CDRASSOC 'R OTHER) T)))
     ($SC-TREE TREE) (SETQ TREE ($GET TREE 'TREE))
     (COND (($ISA-CSA NAME)
            (MESSAGE 'UTIL "$PLANT COMPLAINING: " NAME
                     " IS A CSA OBJECT ALREADY.")
            (RETURN NIL)))
     (SETQ NAME ($NEWOBJ 'SC NAME))
     (MAPC ($FRAGMENT PATTERN NIL)
       (LAMBDA (PART)
         (SETQ LEAF ($PLANT1 ($N-TUPLE-OF (CAR PART)) TREE NIL NIL))
         (SETQ VARS (CDR LEAF))
         (SETQ LEAF (CAR LEAF))
         (COND ((NOT ($IS-SC-LIST LEAF))
                (RPLACA LEAF ($PARENT-OF LEAF))
                (RPLACD LEAF (LIST 'SC))))
         (RPLACD (CDR LEAF)
                 (CONS (LIST NAME
                             (COND ((CDR PART)
                                    (COND ((CDDR PART) (CONS 'AND
                                                             (CDR PART)))
                                          (T (CADR PART))))
                                   (T NIL))
                             (INTO VARS CAR))
```

```
                        ($SCS-OF LEAF)))
        (AUGPROP NAME 'TREE-LOCS LEAF)))
    (EVAL (LIST 'CSETQ NAME BODY))
    ($PUT NAME 'SC-BODY BODY)
    ($PUT NAME 'SC-PATTERN PATTERN)
    ($INTRODUCE NAME) ($TOGGLE NAME)
    ($PUT NAME 'PRIORITY PRIORITY)
    ($PUT NAME 'RUNCONDITION RUNCOND)
    (RETURN NAME)))

(DEFUN $ACTIVATE (PATTERN TREE . WHERE)
  (PROG (LEAVES BINDS TREEBINDS MAPPEDB TAKERS)
    (SETQ WHERE (COND (WHERE (CAR WHERE)) (T 'TAKERS)))
    (SETQ TREE ($GET TREE 'TREE))
    (MAPC ($NIBBLERS PATTERN TREE NIL)
      (LAMBDA (LEAF)
        (SETQ TREEBINDS (CDR LEAF)) (SETQ LEAF (CAR LEAF))
        (MAPC ($CONTEXT-FILTER1 ($SCS-OF LEAF))
          (LAMBDA (D)
            (SETQ MAPPEDB (INTO-N (LAMBDA (X Y) (CONS X (CDR Y)))
                                  (CADDR D) TREEBINDS))
            (SETQ BINDS (COND ((CADR D) ($ALLBINDS ($INSTAN (CADR D)
                                                            MAPPEDB)))
                              (T '(T NIL))))
            (COND (BINDS
                   (MAPC (CDR BINDS)
                         (LAMBDA (B)
                           ($SEND WHERE (LIST (CAR D)
                                              (LIST 'QUOTE
                                                    (APPEND MAPPEDB B)))
                                  ($GET (CAR D) 'PRIORITY)
                                  (EVAL ($GET (CAR D)
                                              'RUNCONDITION)))
                           ))))))))
    (RETURN (EVAL WHERE))))

(DEFUN $INSTAN-D (PATTERN BINDINGS)
  (PROG ((BQUOTE (INTO BINDINGS
                       (LAMBDA (X) (CONS (CAR X) (LIST 'QUOTE (CDR X))))))))
    (RETURN ($INSTAN-D1 PATTERN))))

(DEFUN $INSTAN-D1 (P)
  (COND (($IS-DBQUERY P) ($INSTAN P BINDINGS))
        ((MEMQ (CAR P) '(AND OR ANY))
         (CONS (CAR P) (INTO (CDR P) $INSTAN-D1)))
        (T ($INSTAN P BQUOTE))))

(DEFUN $ZAP (D)
  (MAPC (GET D 'TREE-LOCS)
        (LAMBDA (LEAF) ($WEED D LEAF)))
  ($RECYCLE D))

(DEFUN $SC-TREE (X)
  (COND ((EQ ($TYPE X) 'SC-TREE))
        (($ISA-CSA X) (MESSAGE 'UTIL "$SC-TREE COMPLAINING: " X
                               " IS ALREADY IN USE AS A CSA OBJECT."))
        (T ($NEWOBJ 'SC-TREE X)
           ($PUT X 'TREE ($SPROUT NIL))
           (CSET X (EVAL (SUBST X 'SELF
                                '(LAMBDA (P) (PUT 'SELF 'SIGNAL-BUFFER P)
                                             (MAPC ($ACTIVATE P 'SELF)
                                                   (LAMBDA (D)
                                                     (COND ((EVAL (CDR D))
                                                            (EVAL (CAR D))))))
                                             (GET 'SELF 'SIGNAL-BUFFER)))))))
  X)
```

Appendix B - Channel Code

```
(DEFUN $CHANNEL (CH)
   (COND ((EQ ($TYPE CH) 'CHANNEL))
         (($ISA-CSA CH) (MESSAGE 'UTIL "$CHANNEL COMPLAINING: " CH
                                 " IS AN EXISTING CSA OBJECT."))
         (T ($NEWOBJ 'CHANNEL CH)
            ($PUT CH 'RIGHT CH)
            ($PUT CH 'LEFT CH)
            ($INTRODUCE CH) ($TOGGLE CH)))
   CH)

(DEFUN $CONNECT (PR CH MODE TYPE AT POINT)
   (PROG (TP TAP)
     ($CHANNEL CH)
     (SETQ TAP ($NEWOBJ 'CHANNELTAP NIL))
     (SETQ TP ($FINDTAP CH AT POINT))
     ($PUT TAP 'LEFT TP)
     ($PUT TAP 'RIGHT ($GET TP 'RIGHT))
     ($PUT ($GET TP 'RIGHT) 'LEFT TAP)
     ($PUT TP 'RIGHT TAP)
     ($PUT TAP 'MODE MODE)
     ($PUT TAP 'TYPE TYPE)
     ($PUT TAP 'OBJECT PR)
     ($INTRODUCE TAP) ($TOGGLE TAP)
     (RETURN TAP)))

(DEFUN $FINDTAP (CH AT POINT)
   (COND ((EQ POINT 'RIGHTEND) ($GET CH 'LEFT))
         ((EQ POINT 'LEFTEND) CH)
         (T (WHILE (AND (EQ ($TYPE (SETQ CH ($GET CH 'RIGHT)))
                            'CHANNELTAP)
                        (NEQ ($GET CH 'OBJECT) POINT)))
            (COND ((EQ AT 'BEFORE) ($GET CH 'LEFT))
                  (T CH)))))

(DEFUN $DISCONNECT (PR CH)
   (AND (EQ ($TYPE (SETQ PR ($FINDTAP CH 'AFTER PR))) 'CHANNELTAP)
        ($HIDE PR)))

(DEFUN $KILL-CHANNELTAP (TAP)
   ($PUT ($GET TAP 'LEFT) 'RIGHT ($GET TAP 'RIGHT))
   ($PUT ($GET TAP 'RIGHT) 'LEFT ($GET TAP 'LEFT))
   ($RECYCLE TAP))

(DEFUN $KILL-CHANNEL (CH)
   (WHILE (EQ ($TYPE ($GET CH 'RIGHT)) 'CHANNELTAP)
          ($KILL-CHANNELTAP ($GET CH 'RIGHT)))
   ($RECYCLE CH))

(DEFUN $INJECT (REQ SRVR CH AT START MOVING)
(PROG (RESP AT1) (RETURN
   (COND ((NOT ($VISIBLE CH)))
         (T (SETQ START (SETQ AT ($FINDTAP CH AT START)))
            ($PROPAGATE)
             (COND ((EQ ($TYPE AT) 'CHANNEL) NIL)
                   (T (SETQ RESP (EVAL (LIST ($GET AT 'OBJECT)
                                             (LIST 'QUOTE REQ))))
                      (SETQ AT1 AT) (SETQ SRVR NIL) ($PROPAGATE)
                      (SETQ MOVING (COND ((EQ MOVING 'LEFT) 'RIGHT)
                                         (T 'LEFT)))
                      (WHILE (NEQ (SETQ AT1 ($GET AT1 MOVING)) START)
                        (AND (EQ ($GET AT1 'TYPE) 'RESPONSE-WATCHER)
```

```
                                ($VISIBLE AT1)
                                (DO (SETQ ::T1 (EVAL (LIST ($GET AT1 'OBJECT)
                                                           (LIST 'QUOTE RESP))))
                                    (OR (EQ ($GET AT1 'MODE) 'TRANSPARENT)
                                        (SETQ RESP ::T1)))))
                        RESP)))))))


(DEFUN $PROPAGATE ()
   (WHILE (AND (EQ ($TYPE (SETQ AT ($GET AT MOVING)))
                   'CHANNELTAP)
               (NOT (AND (EQ ($GET AT 'OBJECT) SRVR)
                         ($VISIBLE AT))))
          (AND (EQ ($GET AT 'TYPE) 'WATCHER)
               ($VISIBLE AT)
               (DO (SETQ ::T1 (EVAL (LIST ($GET AT 'OBJECT)
                                          (LIST 'QUOTE REQ))))
                   (OR (EQ ($GET AT 'MODE) 'TRANSPARENT)
                       (SETQ REQ ::T1))))))


(DEFUN $SHOWCHAN FEXPR (CH)
   (EVAL (LIST '$PRINT CH))
   (WHILE (EQ ($TYPE (SETQ CH ($GET CH 'RIGHT))) 'CHANNELTAP)
          (EVAL (LIST '$PRINT CH)))
   T)
```

Appendix C - Database Code

```
(DEFUN $SYMBOL-PATHS (S)
  (CSETQ ::PATHS NIL) ($SYM-PATHS S 0) ::PATHS)

(DEFUN $SYM-PATHS (S PATH)
  (COND ((SCSA-ATOM S)
         (COND ((EQ ($TYPE S) 'DATUM)
                ($SYM-PATHS ($GET S 'NTUPLE) PATH))
               (T (PUSH (CONS S PATH) ::PATHS))))
        ((SETQ PATH (TIMES PATH 10))
         (MAPC-N (LAMBDA (X Y) ($SYM-PATHS X (PLUS PATH Y)))
                 S '(1 2 3 4 5 6 7 8 9)))))

(DEFUN $CANDIDATES (S)
  (CSETQ ::BEST '(NIL 1000000))
  ($MIN-SYM S 0)
  (AND ::BEST (CDDR ::BEST)))

(DEFUN $MIN-SYM (S PATH)
  (COND ((SCSA-ATOM S)
         (COND (($IS-CSA-VAR S))
               (($EQ ($TYPE S) 'DATUM)
                ($MIN-SYM ($GET S 'NTUPLE) PATH))
               ((CSETQ ::T1 (ASSOC PATH ($GET S 'DB-BUCKETS)))
                (COND ((LESSP (CADR ::T1) (CADR ::BEST))
                       (CSETQ ::BEST ::T1))))))
        (T (SETQ PATH (TIMES PATH 10))
           (PROG ((POS 1))
             LOOP ($MIN-SYM (CAR S) (PLUS PATH POS))
                  (COND ((AND ::BEST (SETQ S (CDR S)))
                         (SETQ POS (ADD1 POS)) (GO LOOP))
                        (T (RETURN))))))))

(DEFUN $CREATE-DATUM (D)
  (PROG ((DAT ($NEWOBJ 'DATUM NIL)))
    ($PUT DAT 'NTUPLE D)
    ($INTRODUCE DAT)
    (MAPC ($SYMBOL-PATHS D)
          (LAMBDA (X) (COND ((CSETQ ::T1 (ASSOC (CDR X)
                                                ($GET (CAR X) 'DB-BUCKETS)))
                             (RPLACD (CDR ::T1) (CONS DAT (CDDR ::T1))))
                            (RPLACA (CDR ::T1) (ADD1 (CADR ::T1))))
                           (T (AUGPROP (CAR X) 'DB-BUCKETS
                                       (LIST (CDR X) 1 DAT))))))
    (RETURN DAT)))

(DEFUN $PATTERN-FILTER (TPL CANDS)
  (COLLECT CANDS
      (LAMBDA (C) (COND (($MATCH ($GET C 'NTUPLE) TPL)
                         (CONS C ::BINDS))
                        (T NIL)))))

(DEFUN $MATCH (D TPL)
  (SETQ MATCH-CURRENT-CYCLE (ADD1 MATCH-CURRENT-CYCLE))
  (CSETQ ::BINDS NIL)
  (AND ($MATCH1 D TPL) (CONS T ::BINDS)))

(DEFUN $MATCH1 (D TPL)
  (COND ((ATOM TPL)
         (COND (($IS-CSA-VAR TPL)
                (COND ((EQUAL ($MATCH-CYCLE-NUM TPL) MATCH-CURRENT-CYCLE)
```

```
                        (EQUAL ($MATCH-BINDING-OF TPL) D))
                       (T ($PUT-MATCH-CYCLE TPL .MATCH-CURRENT-CYCLE)
                          ($PUT-MATCH-BINDING TPL D)
                          (PUSH ((CONS TPL D) ::BINDS))))
                 ((EQ ($TYPE TPL) 'DATUM)
                  (OR (EQ TPL D) ($MATCH1 D ($GET TPL 'NTUPLE))))
                 (T (EQ TPL D))))
        ((ATOM D)
         (COND ((EQ ($TYPE D) 'DATUM) ($MATCH1 ($GET D 'NTUPLE) TPL))
               (T NIL)))
        (T (MAPC-N-UNTIL $MATCH1 NULL D TPL))))

(DEFUN $MATCH-CYCLE-NUM (V) (OR (*CDR V) (RPLACD V 0)))

(DEFUN $MATCH-BINDING-OF (V) (EVAL V))

(DEFUN $PUT-MATCH-CYCLE (V C) (RPLACD V C))

(DEFUN $PUT-MATCH-BINDING (V B) (SET V B))


(DEFUN $FETCH (ITEM)
   (COND (W-ACTIVE (WCLEAR 'MEMORY) (WPRINC 'MEMORY ITEM)))
   (CSETQ ::FETCH ($PATTERN-FILTER ITEM ($CONTEXT-FILTER
                                        ($CANDIDATES ITEM))))
   (COND (W-ACTIVE (WCLEAR 'MEMRESP) (WPRINC 'MEMRESP ::FETCH)))
   ::FETCH)


(DEFUN $STORE (ITEM)
   (PROG (D CTX)
      (SETQ D ($PATTERN-FILTER ITEM ($CANDIDATES ITEM)))
      (COND ((SETQ CTX ($CONTEXT-FILTER1 D)) (RETURN (CAAR CTX))))
      (SETQ D (COND (D (CAAR D))
                    (T ($CREATE-DATUM ITEM))))
      ($TOGGLE D)
      (RETURN D)))


(DEFUN $ERASE (ITEM)
   (MAPC ($FETCH ITEM)
         (LAMBDA (D) (AND ($TOGGLE (CAR D)) ($UNCREATE-DATUM (CAR D)))))
   ::FETCH)


(DEFUN $UNCREATE-DATUM (D)
   (MAPC ($SYMBOL-PATHS ($GET D 'NTUPLE))
      (LAMBDA (X)
         (CSETQ ::T1 (ASSOC (CDR X)
                            (CSETQ ::T2 ($GET (CAR X) 'DB-BUCKETS))))
         (DELQ D (CDR ::T1))
         (COND ((CDDR ::T1) (RPLACA (CDR ::T1) (SUB1 (CADR ::T1))))
               (T ($PUT (CAR X) 'DB-BUCKETS (DELQ ::T1 ::T2))))))
   ($RECYCLE D))


(COND ((UNBOUNDP ::DB-ASSIST) (CSETQ ::DB-ASSIST T)))

(COND ((UNBOUNDP MATCH-CURRENT-CYCLE)
       (CSETQ MATCH-CURRENT-CYCLE 0)))

(CSETQ ::DATABASE-INTERFACE-FUNCTIONS '(+ -))

(DEFUN $ALLBINDS (P)
   (COND (($IS-DBQUERY P)
          (COND ((CSETQ ::T1 (EVAL P))
                 (CONS 'T (COND ((CDAR ::T1) (INTO ::T1 CDR))
                                (T '(NIL)))))
                (T NIL)))
         ((EQ (CAR P) 'AND)
```

```
          (PROG ((BINDS '(NIL)) NEXT)
            LOOP (COND ((NULL (SETQ P (CDR P)))
                        (RETURN (CONS 'T BINDS))))
                 (SETQ BINDS (INDEX (INTO BINDS
                                    (LAMBDA (B)
                                      (COND ((SETQ NEXT
                                                   ($ALLBINDS ($INSTAN-D (CAR P) B)))
                                             (INTO (CDR NEXT)
                                                   (LAMBDA (N) (APPEND N B))))
                                            (T NIL))))
                                 NIL APPEND))
                 (COND (BINDS (GO LOOP))
                       (T (RETURN NIL))))
          ((OR (EQ (CAR P) 'OR) (EQ (CAR P) 'ANY))
           (CSETQ ::T1
            (UNIQUE-WRT (INDEX (INTO (CDR P)
                                     (LAMBDA (X) (AND (CSETQ ::T1 ($ALLBINDS X))
                                                      (CDR ::T1))))
                               NIL APPEND)
                        EQUIV))
           (COND (::T1 (CONS 'T ::T1))
                 ((EQ (CAR P) 'ANY) '(T NIL))
                 (T NIL)))
          ((EVAL P) '(T NIL))
          (T NIL)))

(CSETQ ::DATABASE-INTERFACE-FUNCTIONS '(+ -))

(DEFUN + FEXPR (ENERGY ITEM)
  (OR ($FETCH ITEM)
      (AND ::DB-ASSIST (PRINT "DB-ASSIST...") (PRINT ITEM)
           (*READ "RESPONSE: "))))

(DEFUN - FEXPR (ENERGY ITEM)
  (OR ($FETCH ITEM)
      (AND ::DB-ASSIST (PRINT "DB-ASSIST...") (PRINT ITEM)
           (*READ "RESPONSE: "))))

(DEFUN $IS-DBQUERY (X)
  (AND (NOT (ATOM X))
       (MEMQ (CAR X) ::DATABASE-INTERFACE-FUNCTIONS)))
```

Appendix D - Context Code

```
(DEFUN $VISIBLE (ITEM) (CAR ($GET ITEM 'CONTEXT)))

(DEFUN $CONTEXT-FILTER (CANDS)
  (FILTER CANDS (LAMBDA (C) (CAR ($GET C 'CONTEXT)))))

(DEFUN $CONTEXT-FILTER1 (CANDS)
  (FILTER CANDS (LAMBDA (C) (CAR ($GET (CAR C) 'CONTEXT)))))

(DEFUN $POP-CONTEXT ()
  (PROG (CTX)
    (COND ((NULL CONTEXT-STACK) (RETURN NIL)))
    (MAPC (CAR CONTEXT-STACK)
          (LAMBDA (D) (SETQ CTX ($GET D 'CONTEXT))
                      (COND ((CDDR CTX)
                             (RPLACA CTX (NOT (CAR CTX)))
                             (RPLACD CTX (CDDR CTX)))
                            (T ($KILL D))))))
    (SETQ CONTEXT-STACK (CDR CONTEXT-STACK))
    (SETQ CONTEXT-LEVEL (SUB1 CONTEXT-LEVEL))
    (COND (W-ACTIVE (WCLEAR 'CLEVEL) (WPRINC 'CLEVEL CONTEXT-LEVEL)))
    (RETURN CONTEXT-LEVEL)))

(DEFUN $KILL (ITEM)
  (EVAL (LIST (CDRASSOC ($TYPE ITEM)
                        '((DATUM.$UNCREATE-DATUM)
                          (SC.$ZAP)
                          (CHANNEL.$KILL-CHANNEL)
                          (CHANNELTAP.$KILL-CHANNELTAP)))
        (LIST 'QUOTE ITEM))))

(DEFUN $PUSH-CONTEXT ()
  (SETQ CONTEXT-STACK (CONS NIL CONTEXT-STACK))
  (SETQ CONTEXT-LEVEL (ADD1 CONTEXT-LEVEL))
  (COND (W-ACTIVE (WCLEAR 'CLEVEL) (WPRINC 'CLEVEL CONTEXT-LEVEL)))
  CONTEXT-LEVEL)

(DEFUN $RESET-CONTEXT (LEVEL)
  (WHILE (GREATERP CONTEXT-LEVEL LEVEL) ($POP-CONTEXT))
  T)

(DEFUN $INTRODUCE (ITEM) ($PUT ITEM 'CONTEXT (LIST NIL)))

(DEFUN $TOGGLE (ITEM)
  (PROG ((CTX ($GET ITEM 'CONTEXT)))
    (RPLACA CTX (NOT (CAR CTX)))
    (RETURN
      (COND ((AND (CDR CTX) (EQUAL (CADR CTX) CONTEXT-LEVEL))
             (RPLACA CONTEXT-STACK (DELQ ITEM (CAR CONTEXT-STACK)))
             (RPLACD CTX (CDDR CTX))
             (NOT (CDR CTX)))
            (T (RPLACA CONTEXT-STACK (CONS ITEM (CAR CONTEXT-STACK)))
               (RPLACD CTX (CONS CONTEXT-LEVEL (CDR CTX)))
               NIL)))))

(DEFUN $HIDE (OBJ)
  (AND ($VISIBLE OBJ) ($TOGGLE OBJ) ($KILL OBJ)))

(DEFUN $UNHIDE (OBJ) (OR ($VISIBLE OBJ) ($TOGGLE OBJ)))
```

Appendix E - Miscellaneous CSA SC-Related Code

```
(COND ((UNBOUNDP ::AGENDA) (CSETQ ::AGENDA NIL)))

(DEFUN $SEND (DEST ITEM PRIOR . RUNCOND)
  (SETQ ITEM (CONS ITEM (COND (RUNCOND (CAR RUNCOND)) (T))))
  (SET DEST
    (COND ((EQ PRIOR 'FRONT) (CONS ITEM (EVAL DEST)))
          ((EQ PRIOR 'REAR) (APPEND (EVAL DEST) (CONS ITEM NIL)))
          (T (CONS ITEM (EVAL DEST))))))

(DEFUN $MAKE-CSA-VAR (A)
  (CSETQ ::T1 (COND ((ATOM A) (EXPLODE A)) (T A)))
  (COND ((NEQ (CAR ::T1) ::CSA-VAR-SYM)
         (COMPRESS (CONS ::CSA-VAR-SYM ::T1)))
        ((ATOM A) A)
        (T (COMPRESS A))))

(CSETQ ::CSA-VAR-SYM '-)

(DEFUN $IS-CSA-VAR (A) (EQ (CAR (EXPLODE A)) ::CSA-VAR-SYM))

 (DEFUN $INIT ()
   (MAPC '(CSA-OBJ-TYPES CSA-NAMED CSA-NAMELESS)
     (LAMBDA (X) (CSET X NIL)))
   (COND ((UNBOUNDP RECYCLED-NODES)
          (CSETQ RECYCLED-NODES NIL)))
   (CSETQ CONTEXT-LEVEL 0)
   (CSETQ  CONTEXT-STACK '(NIL))
   (CSETQ CURRENT-ACTIVITY 'PERCEIVING-REALITY)
 ;DEFINE THE CSA MEMORY NODE TYPES
   (MAPC '(
     (SC (SC-PATTERN SC-BODY TREE-LOCS
          CONTEXT PRIORITY RUNCONDITION) (TREE-LOCS))
     (SC-TREE (TREE SIGNAL-BUFFER) (TREE))
     (CHANNEL (LEFT RIGHT CONTEXT))
     (CHANNELTAP (OBJECT MODE TYPE LEFT RIGHT CONTEXT))
     (ABS-MECH (NAME PARTS EVENTS PURPOSE TRIGGER INITIAL-WORLD))
     (ABS-MECH-EVENT (PART-OF CLASS NTUPLE LINKS))
     (ABS-OBJ (NAME PHYS-DESC PART-OF CONCEPT-FRAME))
     (SYNTAX-NODE (MEANING REFERENT))
     (ABS-MECH-LINK (TYPE INSTANCE-OF EVENTS GATES))
     (CSA-PRIM (CLASS ARGCOUNT NETS OCCURS-IN-DATA OCCURS-IN-ALGS
                OCCURS-IN-INDUCEMENTS OCCURS-IN-PREDICTIONS
                OCCURS-IN-SEMANTICS))
     (LEX-WORD (NETS CSA-MARKERS MEANING))
     (CSA-LINK (EVENT-TYPES GATE-TYPES))
     (NET-TEST (TEST VARS OCCURS-IN))
     (NET-NODE (BYPASSES TEST TESTTYPE BINDINGS CHOICES OCCURS-IN ACTION))
     (ABS-ALG (ACCOMPLISHES EVENTS VARIABLES OCCURS-IN THINGS AGENT))
     (ABS-ALG-EVENT (PART-OF CLASS NTUPLE LINKS RECOMMENDATION))
     (ABS-ALG-LINK (TYPE INSTANCE-OF EVENTS GATES))
     (DESCRIPTOR (PATTERN VARIABLE OCCURS-IN))
     (TOKEN ())    ? SPACE RESERVED FOR "TOKEN" ENTITY
     (DATUM (NTUPLE CONTEXT))
     (SYN-ALG (ACCOMPLISHES EVENTS INSTANCE-OF AGENT))
     (SYN-ALG-EVENT (PART-OF CLASS NTUPLE LINKS))
     (SYN-ALG-LINK (TYPE INSTANCE-OF EVENTS GATES))
     (ALG-LINKER (ALG BINDINGS OCCURS-IN RECOMMENDATION))
     (INDUCEMENT (NTUPLES OCCURS-IN))
     (PREDICTION (NTUPLES OCCURS-IN))
     (MEANING (NTUPLES OCCURS-IN))
     (NETWORK (TYPE TOPNODE VARS OCCURS-IN))
     (CTX-NODE (OCCURS-IN CHOICES LINKERS))
```

```
      (CTX-LINKER (OCCURS-IN FUNC STACKLENGTH))
    ) (LAMBDA (X) (PUT (CAR X) 'SLOTSTRUCTURE (CADR X))
                  (COND ((CDDR X) (PUT (CAR X) 'IGNORESLOTS (CADDR X))))
                  (CSETQ CSA-OBJ-TYPES (CONS (CAR X) CSA-OBJ-TYPES))
                  (CSET (CAR X) NIL)))
 ?DEFINE THE CSA ALGORITHMIC LINK SEMANTICS
    (MAPC '(
      (C-CAUSE ((A T) (S SC)) (S)) ?CONTINUOUS CAUSE
      (R-CAUSE ((A T) (S SC)) (S)) ?REPETITIVE CAUSE
      (C-DENY ((A T) (S SC)) (S))
      (OS-DENY ((A T) (S SC)) (S))
      (OS-CAUSE ((A T) (S)) (S))              ?ONE-SHOT CAUSE
      (C-ENABLE ((S) (A T)) NIL)
      (OS-ENABLE ((S) (A T)) NIL)
      (S-COUPLE ((S SC) (S SC)) (S))
      (THRESH ((SC) (S)) NIL)
      (A-MOTIV ((S SC) (A)) (S))              ?ALGORITHMIC MOTIVATION
      (C-MOTIV ((S SC) (A)) (S))              ?COMPENSATORY MOTIVATION
      (C-BYPROD ((A T) (S SC)) (S))
      (R-BYPROD ((A T) (S SC)) (S))
      (OS-BYPROD ((A T) (S)) (S))
      (ANTAG ((S) (S)) NIL)
      (M-ANTAG ((S) (S) (A)) NIL)
      (INDUCE ((A S SC T) (S SC)) (S))
    ) (LAMBDA (X) ($NEWOBJ 'CSA-LINK (CAR X))
                  ($PUT (CAR X) 'EVENT-TYPES (CADR X))
                  ($PUT (CAR X) 'GATE-TYPES (CADDR X))
                  ($RECORD (CAR X))))
     "INITIALIZED")

(DEFUN $CSA-ATOM (A) (ATOM A))

(CSETQ $PUT PUT)

(CSETQ $GET GET)

(DEFUN $TYPE (OBJ)
   (COND ((AND ($CSA-ATOM OBJ) (GET OBJ 'CSA)) (CAR (GET OBJ 'CSA)))
         (T 'UNKNOWN)))

(DEFUN $NEWOBJ (TYPE . NAME)
   (PROG (OBJ)
      (COND (NAME (SETQ NAME (CAR NAME))))
      (SETQ OBJ (COND ((NULL NAME)
                       (COND (RECYCLED-NODES
                              (PROG (X) (SETQ X (CAR RECYCLED-NODES))
                                        (CSETQ RECYCLED-NODES
                                               (CDR RECYCLED-NODES))
                                        (CSETQ CSA-NAMELESS (CONS X
                                                                  CSA-NAMELESS))
                                        (RETURN X)))
                             (T (CSETQ CSA-NAMELESS
                                       (CONS (INTERN (GENSYM))
                                             CSA-NAMELESS))
                                (CAR CSA-NAMELESS))))
                      (T (CSETQ CSA-NAMED (CONS NAME CSA-NAMED))
                         NAME)))
      (COND ((NOT (MEMBER TYPE CSA-OBJ-TYPES))
             (MESSAGE 'UTIL "$NEWOBJ COMPLAINING: UNKNOWN OBJECT TYPE " TYPE)))
      (PUT OBJ 'CSA (LIST TYPE))
      (RETURN OBJ)))

(DEFUN $RECYCLE (NODE)
   (REMOB NODE)
   (COND ((MEMQ NODE CSA-NAMELESS)
          (CSETQ RECYCLED-NODES (CONS NODE RECYCLED-NODES))
```

```
            ((SETQ CSA-NAMELESS (DELQ NODE CSA-NAMELESS)))
           (T ((SETQ CSA-NAMED (DELQ NODE CSA-NAMED))))
    NODE)

(DEFUN $INSTAN (STRUC SUBLST)
   (PROG ()
     LOOP  (COND ((NULL SUBLST) (RETURN STRUC)))
           (SETQ STRUC (SUBST (CDAR SUBLST) (CAAR SUBLST) STRUC))
           (SETQ SUBLST (CDR SUBLST))
           (GO LOOP)))

(DEFUN $RECORD (MEM-OBJ)
   (COND ((NOT (MEMBER MEM-OBJ (EVAL ($TYPE MEM-OBJ))))
          (SET ($TYPE MEM-OBJ) (CONS MEM-OBJ (EVAL ($TYPE MEM-OBJ)))))))
   MEM-OBJ)


(DEFUN $P MACRO OBJS (CONS '$PRINT OBJS))

(DEFUN $ISA-CSA (X) (AND (OR (ATOM X) ($CSA-ATOM X))
                         (GET X 'CSA)))

(CSETQ MESSAGE (LAMBDA TEXT
        (PROG ()  (INTO (CDR TEXT) (LAMBDA (Y) (PRIN1 Y)))
                  (TERPRI) )))

(DEFUN $PRINT FEXPR OBJS
   (MAPC OBJS (LAMBDA (OBJ)
   (PROG (OBJTYPE PROPLST SLOTNAMES IGNORE)
   (SETQ OBJTYPE ($TYPE OBJ))
   (SETQ PROPLST (*CDR OBJ))
   (SETQ SLOTNAMES (GET OBJTYPE 'SLOTSTRUCTURE))
   (SETQ IGNORE (GET OBJTYPE 'IGNORESLOTS))
     (MESSAGE 'UTIL " ")
     (MESSAGE 'UTIL "**OBJECT " OBJ " (" OBJTYPE ")**")
    LOOP
    (COND ((NULL SLOTNAMES) NIL)
          (T
       (MESSAGE 'UTIL "    " (CAR SLOTNAMES) ":  "
                (COND ((MEMBER (CAR SLOTNAMES) IGNORE) '**IGNORED**)
                      (T ($GET OBJ (CAR SLOTNAMES)))))
       (SETQ SLOTNAMES (CDR SLOTNAMES))
          (GO LOOP)))
     (PROG () LOOP (COND ((NULL PROPLST) (RETURN NIL))
                   ((AND (NEQ (CAAR PROPLST) 'CSA)
                         (NOT (MEMBER (CAAR PROPLST)
                                      (GET OBJTYPE 'SLOTSTRUCTURE))))
                    (MESSAGE 'UTIL "       " (CAAR PROPLST) ": "
                             (CDAR PROPLST))))
           (SETQ PROPLST (CDR PROPLST)) (GO LOOP))
     (WINC 'UTIL 1) (WINC 'UTIL 1)
     (RETURN NIL)))))

(csetq winc (lambda x nil))
```

Appendix F - LISP Extensions and Incompatibilities

```
(CSETQ ::DEFUNNED NIL)

(DEFMAC DEFUN (LAMBDA ARG
        (PROG (TYPE) (COND ((SETQ TYPE (ASSOC (CADR ARG)
                                    '((MACRO . DEFMAC)(FEXPR . DEFSPEC)(EXPR . CSETQ))
                                    )) (RPLACD ARG (CDDR ARG)))
                           (T (SETQ TYPE '(EXPR . CSETQ))))
   (COND ((NOT (MEMBER (CAR ARG) ::DEFUNNED))
          (CSETQ ::DEFUNNED (CONS
                  (CAR ARG)
                  ::DEFUNNED))))
     ((COND ((NOT (OR (EQ (*CAR 'SOURCE-ELT) (*CAR '!,!,!.!,))
                      (EQ SOURCE-ELT 'UNKNOWN)))
            (PUT (CAR ARG) 'SOURCE-ELT SOURCE-ELT)))
      (SETQ TYPE (CDR TYPE))
          (RETURN (LIST TYPE (CAR ARG)(CONS 'LAMBDA (CDR ARG)))))))

(DEFUN CDRASSOC (X Y) (COND ((CSETQ ::T1 (ASSOC X Y)) (CDR ::T1))
                            (T NIL)))


(DEFUN INTO-N (!,FN . !,LISTS)
   (PROG (RESULT)
     LOOP (COND ((FIND-FIRST !,LISTS ATOM) (RETURN (REVERSE RESULT))))
          (SETQ RESULT (CONS (EVAL (CONS !,FN
                                   (INTO !,LISTS (LAMBDA (X) (LIST 'QUOTE (CAR X))))))
                             RESULT))
          (SETQ !,LISTS (INTO !,LISTS CDR))
          (GO LOOP)))


(DEFUN MAPC-N (!,FN . !,LISTS)
   (PROG ()
     LOOP (COND ((FIND-FIRST !,LISTS ATOM) (RETURN !,LISTS)))
          (EVAL (CONS !,FN
                      (INTO !,LISTS (LAMBDA (X) (LIST 'QUOTE (CAR X))))))
          (SETQ !,LISTS (INTO !,LISTS CDR))
          (GO LOOP)))


(DEFUN MAPC-N-UNTIL (!,FN !,CRIT . !,LISTS)
   (PROG ()
     LOOP (COND ((FIND-FIRST !,LISTS ATOM) (RETURN !,LISTS)))
          (COND ((!,CRIT (EVAL (CONS !,FN
                               (INTO !,LISTS (LAMBDA (X) (LIST 'QUOTE (CAR X)))))))
                 (RETURN NIL)))
          (SETQ !,LISTS (INTO !,LISTS CDR))
          (GO LOOP)))


(DEFUN MEMQ (X Y)
   (COND ((NULL Y) NIL)
         ((EQ X (CAR Y)) Y)
         (T (MEMQ X (CDR Y)))))


(DEFUN PUSH FEXPR (!,ITEM !,VAR)
   (SET !,VAR (CONS (EVAL !,ITEM) (EVAL !,VAR))))


(DEFUN COLLECT (LST FN)
   (PROG (RESULT TEMP)
     LOOP (COND ((NULL LST) (RETURN (REVERSE RESULT)))
                ((SETQ TEMP (FN (CAR LST)))
                 (SETQ RESULT (CONS TEMP RESULT))))
          (SETQ LST (CDR LST)) (GO LOOP)))
```

```
(DEFUN FILTER (LST FN)
  (PROG (RESULT)
    LOOP (COND ((NULL LST) (RETURN (REVERSE RESULT)))
               ((FN (CAR LST)) (SETQ RESULT (CONS (CAR LST) RESULT))))
         (SETQ LST (CDR LST)) (GO LOOP)))

(DEFUN DEF FEXPR (FN)
  (COND ((*DEF FN))
        ((*DEF (RPLACA 'QQ (*SPEC FN))))
        ((*DEF (RPLACA 'QQ (*MACRO FN))))
        (T 'UNDEFINED)))

(DEFUN UNBOUNDP FEXPR (A) (EQ (*CAR A) (*CAR '#,#,#,#.)))

(DEFUN MAKEUNBOUND (A) (RPLACA A (*CAR '#,#,#,#.)) A)

(DEFUN DELQ (ITM LST)
  (CSETQ ::DONE NIL)
  (COND ((ATOM LST) NIL)
        ((EQ (CAR LST) ITM) (CSETQ ::DONE T) (CDR LST))
        (T (PROG ((L LST))
             LOOP (COND ((NULL (CDR L)) (RETURN LST))
                        ((EQ (CADR L) ITM) (RPLACD L (CDDR L))
                         (CSETQ ::DONE T) (RETURN LST))
                        (T (SETQ L (CDR L)) (GO LOOP)))))))

(DEFUN RASSOC-DELQ (ITM LST)
  (CSETQ ::DONE NIL)
  (COND ((ATOM LST) NIL)
        ((EQ (CDAR LST) ITM) (CSETQ ::DONE T) (CDR LST))
        (T (PROG ((L LST))
             LOOP (COND ((ATOM (CDR L)) (RETURN LST))
                        ((EQ (CDADR L) ITM) (RPLACD L (CDDR L))
                         (CSETQ ::DONE T) (RETURN LST))
                        (T (SETQ L (CDR L)) (GO LOOP)))))))

(DEFUN ASSOC-DELQ (ITM LST)
  (CSETQ ::DONE NIL)
  (COND ((ATOM LST) NIL)
        ((EQ (CAAR LST) ITM) (CSETQ ::DONE T) (CDR LST))
        (T (PROG ((L LST))
             LOOP (COND ((ATOM (CDR L)) (RETURN LST))
                        ((EQ (CAADR L) ITM) (RPLACD L (CDDR L))
                         (CSETQ ::DONE T) (RETURN LST))
                        (T (SETQ L (CDR L)) (GO LOOP)))))))

(DEFUN DELETE (ITM LST)
  (CSETQ ::DONE NIL)
  (COND ((ATOM LST) NIL)
        ((EQUAL (CAR LST) ITM) (CSETQ ::DONE T) (CDR LST))
        (T (PROG ((L LST))
             LOOP (COND ((NULL (CDR L)) (RETURN LST))
                        ((EQUAL (CADR L) ITM) (RPLACD L (CDDR L))
                         (CSETQ ::DONE T) (RETURN LST))
                        (T (SETQ L (CDR L)) (GO LOOP)))))))

(DEFUN DO-LOOP MACRO (VARS EXIT . BODY)
  (PROG (INCS) (RETURN
    (APPEND (LIST 'PROG (APPEND (INTO VARS (LAMBDA (V)
                                             (LIST (CAR V) (CADR V))))
                                (SETQ INCS (INTO VARS (LAMBDA (V)
```

```
                                                    (LIST (INTERN (GENSYM))
                                                      (COND ((CDDR V) (CADDR V))
                                                            (T 1)))))))
                          'LOOP
                          (LIST 'COND (LIST (CAR EXIT) (LIST 'RETURN
                                (COND ((CDR EXIT)
                                       (COND ((CDDR EXIT)
                                              (CONS 'DO (CDR EXIT)))
                                             (T (CADR EXIT))))
                                      (T NIL))))))
                    (APPEND BODY
                      (APPEND (INTO-N
                                (LAMBDA (V I) (LIST 'SETQ (CAR V)
                                                    (LIST 'PLUS (CAR V) (CAR I))))
                                VARS INCS)
                              '((GO LOOP)))))))

(DEFUN INTERN (A) (COMPRESS (EXPLODE A)))

(DEFUN FIND-FIRST (LST CRIT)
  (PROG () LOOP (COND ((ATOM LST) (RETURN NIL))
                      ((CRIT (CAR LST)) (RETURN LST))
                      (T (SETQ LST (CDR LST)) (GO LOOP)))))

(DEFUN INTO (LST FN)
  (PROG (RESULT)
    LOOP (COND ((ATOM LST) (RETURN (REVERSE RESULT))))
         (SETQ RESULT (CONS (FN (CAR LST)) RESULT))
         (SETQ LST (CDR LST)) (GO LOOP)))

(DEFUN MAPC (LST FN)
  (PROG ()
    LOOP (COND ((ATOM LST) (RETURN NIL)))
         (FN (CAR LST))
         (SETQ LST (CDR LST)) (GO LOOP)))

(DEFUN RASSOC (ITM LST)
  (PROG ()
    LOOP (COND ((ATOM LST) (RETURN NIL))
               ((EQ ITM (CDAR LST)) (RETURN (CAR LST))))
         (SETQ LST (CDR LST)) (GO LOOP)))

(DEFUN PUTPROP MACRO (X Z Y) (LIST 'PUT X Y Z))

(DEFUN IMPLODE MACRO (X) (LIST 'COMPRESS X))

(DEFUN PLIST MACRO (X) (LIST '*CDR X))

(DEFUN REMOB (A)
  (MAKEUNBOUND A)
  (RPLACD A NIL))

(DEFUN ALLBUT (ITM LST)
  (COND ((ATOM LST) LST)
        ((EQ ITM (CAR LST)) (CDR LST))
        (T (CONS (CAR LST) (ALLBUT ITM (CDR LST))))))

(DEFUN NEQ MACRO (X Y) (LIST 'NOT (LIST 'EQ X Y)))
```

```
(DEFUN NEQUAL MACRO (X Y) (LIST 'NOT (LIST 'EQUAL X Y)))

(DEFUN AUGPROP (ATM PRP ITM)
  (PUT ATM PRP (CONS ITM (GET ATM PRP))))

(DEFUN UNIQUE (LST)
   (COND ((NULL LST) NIL)
         ((MEMBER (CAR LST) (CDR LST)) (UNIQUE (CDR LST)))
         (T (CONS (CAR LST) (UNIQUE (CDR LST))))))

(DEFUN INSTAN (STRUC SUBLST)
   (PROG ()
     LOOP  (COND ((NULL SUBLST) (RETURN STRUC)))
           (SETQ STRUC (SUBST (CDAR SUBLST) (CAAR SUBLST) STRUC))
           (SETQ SUBLST (CDR SUBLST))
           (GO LOOP)))

(DEFUN EQUIV (PAT1 PAT2)
   (COND ((NEQUAL (LENGTH PAT1) (LENGTH PAT2)) NIL)
   (T (PROG (PAT1T) (SETQ PAT1T PAT1)
      LOOP1 (COND ((NULL PAT1) (GO LOOP2))
                  ((MEMBER (CAR PAT1) PAT2)
                   (SETQ PAT1 (CDR PAT1)) (GO LOOP1))
                  (T (RETURN NIL)))
      LOOP2 (COND ((NULL PAT2) (RETURN T))
                  ((MEMBER (CAR PAT2) PAT1T)
                   (SETQ PAT2 (CDR PAT2)) (GO LOOP2))
                  (T (RETURN NIL)))))))

(DEFUN UNIQUE-WRT (LST CRIT)
   (COND ((ATOM LST) LST)
         ((SATISFIES (CAR LST) (CDR LST) CRIT)
          (UNIQUE-WRT (CDR LST) CRIT))
         (T (CONS (CAR LST) (UNIQUE-WRT (CDR LST) CRIT)))))

(DEFUN SATISFIES (ITEM LST CRIT)
   (PROG ()
    LOOP (COND ((NULL LST) (RETURN NIL))
         ((CRIT ITEM (CAR LST)) (RETURN (CAR LST)))
         (T (SETQ LST (CDR LST)) (GO LOOP)))))

(DEFUN SUBSET (SET1 SET2)   ?  SET1 SUSET OF SET2?
  (PROG () LOOP (COND ((NOT (AND SET1 (MEMBER (CAR SET1) SET2)))
                       (RETURN (NULL SET1))))
                (SETQ SET1 (CDR SET1))  (GO LOOP) ))

(DEFUN NTAIL (N LST)   ?  RETURN LAST N ELTS OF LST
  (PROG () LOOP (COND ((ZEROP N) (RETURN LST))
                (T (SETQ N (SUB1 N)) (SETQ LST (CDR LST)) (GO LOOP)))))

(DEFUN OCCURS-IN (ITM STR)   ?  RECURSIVE [MEMBER] FNCN
   (COND ((ATOM STR) (EQUAL ITM STR))
         (T (OR (OCCURS-IN ITM (CAR STR)) (OCCURS-IN ITM (CDR STR))))))

(DEFUN SUBTRACT (SET1 SET2)   ? SET OPERATION: SET2-SET1
   (COND ((NULL SET2) NIL)
         ((MEMBER (CAR SET2) SET1) (SUBTRACT SET1 (CDR SET2)))
         (T (CONS (CAR SET2) (SUBTRACT SET1 (CDR SET2))))))
```

```
(DEFUN BREADTH-FIRST (LST FCT DEPTH)
   (PROG (RESULT NEXT)
         (WHILE (AND (GREATERP DEPTH 0)
                     (SETQ NEXT (INDEX (INTO LST FCT) NIL APPEND)))
                (PROGN (SETQ DEPTH (SUB1 DEPTH))
                       (SETQ RESULT (APPEND RESULT NEXT))
                       (SETQ LST NEXT)))
         (RETURN RESULT)))

(DEFUN NEWSYM () (INTERN (GENSYM)))

(DEFUN UNIFY (TEST1 VARS1 TEST2 VARS2)
   (COND ((NEQUAL (LENGTH TEST1) (LENGTH TEST2)) NIL)
   (T (PROG (BINDS TEMP1 TEMP2) (GO LOOP1)
       LOOP (SETQ TEST1 (CDR TEST1)) (SETQ TEST2 (CDR TEST2))
       LOOP1
       (COND ((NULL TEST1) (RETURN (LIST BINDS)))
             ((MEMBER (CAR TEST1) VARS1)
              (COND ((MEMBER (CAR TEST2) VARS2)
                     (SETQ TEMP1 (ASSOC (CAR TEST1) BINDS))
                     (SETQ TEMP2 (ASSOC2 (CAR TEST2) BINDS))
                     (COND ((AND (NULL TEMP1) (NULL TEMP2))
                            (SETQ BINDS (CONS (CONS (CAR TEST1)
                                                    (CAR TEST2)) BINDS))
                            (GO LOOP))
                           ((EQ TEMP1 TEMP2) (GO LOOP))
                           (T (RETURN NIL))))
                    (T (RETURN NIL))))
             ((OR (NEQUAL (CAR TEST1) (CAR TEST2))
                  (MEMBER (CAR TEST2) VARS2))
              (RETURN NIL))
             (T (GO LOOP)))))))

(DEFUN ASSOC2 (ITM LST)
   (PROG () LOOP (COND ((NULL LST) (RETURN NIL))
                       ((EQUAL ITM (CDAR LST)) (RETURN (CAR LST)))
                       (T (SETQ LST (CDR LST)) (GO LOOP)))))

(DEFUN WHILE MACRO (PRED . BODY)
   (LIST 'PROG NIL 'LOOP
     (LIST 'COND (LIST PRED (STACK BODY) '(GO LOOP))
                 '(T (RETURN)))))

(DEFUN TREESIZE (X)
   (COND ((ATOM X) 0)
         (T (PLUS 1 (TREESIZE (CAR X)) (TREESIZE (CDR X))))))

(DEFUN GRAPHSIZE (GRAPH) (GRAPHSIZE1 GRAPH NIL))
(DEFUN GRAPHSIZE1 (G SEEN)
   (COND ((OR (ATOM G) (MEMQ G SEEN)) 0)
         (T (PLUS 1 (GRAPHSIZE1 (CAR G) (CONS G SEEN))
                    (GRAPHSIZE1 (CDR G) (CONS G SEEN))))))

(DEFUN INDENT (N)
   (PROG () LOOP (COND ((LESSP (SETQ N (SUB1 N)) 0) (RETURN NIL)))
                 (PRIN1 '! ) (GO LOOP)))

(DEFUN NONIL (LST)
   (COND ((NULL LST) NIL)
         ((ATOM LST) LST)
         ((NULL (CAR LST)) (NONIL (CDR LST)))
```

```
        (T (APPEND (LIST (CAR LST)) (NONIL (CDR LST))))))

(DEFUN SQUASH MACRO (LST)
   (LIST 'INDEX LST NIL 'APPEND))

(DEFUN DAYTIME ()
   (PROG (FRAC SEC MINU HOUR TIM)
      (SETQ TIM (DTIME))
      (SETQ FRAC (REMAINDER TIM 1000))
      (SETQ SEC (QUOTIENT TIM 1000))
      (SETQ MINU (QUOTIENT SEC 60))
      (SETQ SEC (REMAINDER SEC 60))
      (SETQ HOUR (QUOTIENT MINU 60))
      (SETQ MINU (REMAINDER MINU 60))
      (PRIN1 HOUR)
      (SETQ MINU (STRING (COMPRESS (CONS ': (EXPLODE MINU)))))
      (PRIN1 MINU)
      (SETQ SEC (STRING (COMPRESS (CONS ': (EXPLODE SEC)))))
      (PRIN1 SEC)
      (PRIN1 (COMPRESS (CONS '. (EXPLODE FRAC))))
      (TERPRI)))

(DEFUN MAKE-ALIST (VARS VALS)
   (COND ((NULL VALS) VARS) ((NULL VARS) VALS)
         (T (CONS (CONS (CAR VARS) (CAR VALS))
                  (MAKE-ALIST (CDR VARS) (CDR VALS))))))
```